Python Problem Sets

by

Sean Strout and John Sweeney

Purpose

Students are eager to learn computer programming in the python language. First released in 1991 python, the computer programming language, was designed by Guido van Rossum who sets the standards for python computer programs. He encouraged the development of computer programs that are easily read and use the most apparent programming techniques. The python community supports that emphasis, so the extensive collection of support materials, such as introductory books, and advanced python modules continue the tradition of ease of use. Students benefit both from the ease of learning a relatively small computer language and from the ease of access to an extensive library of support materials. A simple python program such as **print(4)** will display **4** on the computer monitor. Such brevity and clarity together with a small set of language commands has lead to python becoming a favorite introductory language for students learning to program computers.

This free open-source computer programming language is available for download from https://www.python.org/.

To maintain student motivation each chapter includes problems sets organized by academic major or concentration. Where possible students should strive to build computer-based tools, or apps, to help them complete projects in their own discipline.

Chapter One – Your first Python program

Installing Python on your own computer

Select the appropriate module for your computer from the dowload list at http://www.python.com. This book is based on python 3.4. Many books and computer modules still rely on version 2.7. Most commands have the same format in both versions. The print command for version 3.4 uses parentheses, but version 2.7 does not.

First Python Computer Program

Open the IDLE module. IDLE's prompt is the chevron, >>>. Type your first command **print('hello world')** IDLE will display **hello world** on the next line. A string is a list of characters, such as letters, symbols, or digits. String delimiters are either a pair of single quotation marks, or a set of double quotation marks surrounding the string. The print command displays “hello world”.

Chapter 2: Getting started with Python Functions

Purpose

Repetitive tasks are a bore, so python, the computer programming language, includes functions, which can be repeatedly called upon to perform such repetitive tasks. However, repeating the same lines in songs such as Scarborough Fair can be fun.

Notice lines two and four are repeated in each stanza. That repetition explains why this song was selected as motivation for our next Python programming project.

Are you going to Scarborough Fair?
Parsley, sage, rosemary, and thyme;
Remember me to the one who lives there,
For once she was a true love of mine.

Tell her to make me a cambric shirt,
Parsley, sage, rosemary, and thyme;
Sewn without seams or fine needlework,
If she would be a true love of mine.

Tell her to wash it in yonder well,
Parsley, sage, rosemary, and thyme;
Where never spring water or rain ever fell,
And she shall be a true lover of mine.

Tell her to dry it on yonder thorn,
Parsley, sage, rosemary, and thyme;
Which never bore blossom since Adam was born,
Then she shall be a true lover of mine.

Now he has asked me questions three,
Parsley, sage, rosemary, and thyme;
I hope he'll answer as many for me
Before he shall be a true lover of mine.

Tell him to buy me an acre of land,
Parsley, sage, rosemary, and thyme;

Between the salt water and the sea sand,
Then he shall be a true lover of mine.

Tell him to plough it with a ram's horn,
Parsley, sage, rosemary, and thyme;
And sow it all over with one pepper corn,
And he shall be a true lover of mine.

Tell him to sheer't with a sickle of leather,
Parsley, sage, rosemary, and thyme;
And bind it up with a peacock feather.
And he shall be a true lover of mine.

Tell him to thrash it on yonder wall,
Parsley, sage, rosemary, and thyme,
And never let one corn of it fall,
Then he shall be a true lover of mine.

When he has done and finished his work.
Parsley, sage, rosemary, and thyme:
Oh, tell him to come and he'll have his shirt,
And he shall be a true lover of mine.[1]

Your new function is named scarborough, not Scarborough, because names starting with a capital letter are retained for special meanings which will be explained later. The function's defining line with the def command, the name of the function, and parentheses for inclusion of a list of arguments. No such arguments are needed for this function so the list is empty and is represented simply by a matching pair of parentheses.

You might begin with a series of print statements that display the first stanza.

```
>>> def scarborough():

print("Are you going to Scarborough Fair?")

print("Parsley, sage, rosemary, and thyme,")

print("Remember me to the one who lives there,")
print("For once she was a true love of mine.")
```

Each command is indented the same amount. Thus Python knows those lines belong to the same function.

To run, or call, your new function you simply type the name of your function following the input prompt which is a chevron. Because scarborough is a function you must include the parentheses for a list of arguments event though this function does not expect any such list.

```
>>> scarborough()

Are you going to Scarborough Fair?

Parsley, sage, rosemary, and thyme,

Remember me to the one who lives there,

For once she was a true love of mine.

>>>
```

Notice the function begins with the def command. Also that the function name, scarborough, is followed by parentheses (). Those parentheses allow the function to accept a list of arguments. Later example python programs will demonstrate that capability.

When you include the second stanza, you will notice lines two are repeated in each stanza.

```
>>> def scarborough():

print("Are you going to Scarborough Fair?")

print("Parsley, sage, rosemary, and thyme,")

print("Remember me to the one who lives there,")

print("For once she was a true love of mine.")

print()

print("Tell her to make me a cambric shirt,")

print("Parsley, sage, rosemary, and thyme,")

print("Sewn without seams or fine needlework,")

print("If she would be a true love of mine.")

print()


>>> scarborough()

Are you going to Scarborough Fair?

Parsley, sage, rosemary, and thyme,

Remember me to the one who lives there,

For once she was a true love of mine.


Tell her to make me a cambric shirt,

Parsley, sage, rosemary, and thyme,
```

Sewn without seams or fine needlework,

If she would be a true love of mine.

```
>>>
```

Python functions can call other functions. The following python code displays only the first stanza.

```
>>> def herbs():

print("Parsley, sage, rosemary, and thyme,")


>>> def scarborough():

print("Are you going to Scarborough Fair?")

herbs()

print("Remember me to the one who lives there,")

print("For once she was a true love of mine.")


>>> scarborough()

Are you going to Scarborough Fair?

Parsley, sage, rosemary, and thyme,

Remember me to the one who lives there,
For once she was a true love of mine.
>>>
```

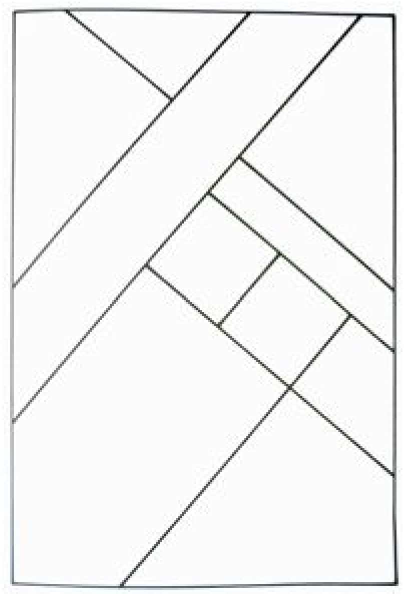
This technique can be reproduced in each of the following stanzas thus reducing the work of typing your Python computer program.

Examine the fourth line of each stanza. Beginning with the third stanza those lines are very similar to each other. The only difference is how those lines begin. Again use def and the new function's name. Include the variable start in the list of arguments. The the print command uses the + operator to join together the two strings: one in the start variable and the other string between the double quotation marks.

```
>>> def lover(start):
print(start + "shall be a true lover of mine")


>>> lover("Then she ")
Then she shall be a true lover of mine
>>> lover("Before he ")
Before he shall be a true lover of mine
>>>
```

Using different phrases for the start of each line allows this new function to produce a restricted variety of lines.

Art problem:



You must calculate the area of red glass for a stained glass window. The red areas area a square of size 2, a rectangle of size 1 by 4, a square of size 3, and a rectangle of size 2 by 3.

Hint: Develop a function to compute the area of a square. The area of a square is the edge length times itself. Develop a function to compute the area of a rectangle. The area of a rectangle is length times width. The total area is the sum of those four areas.

Possible solution:

```
""" Return area of a square given the length of an edge """

def square(edge):

return edge * edge
```

```
""" Return area of a square given the length of an edge """

def rectangle(length, width):

return length * width


""" Calculate total area of red colored glass """

def red():

total = square(2) + rectangle(1, 4) + square(3) + rectangle(2, 3)

return total


print("total area of red glass: ", red())
```

Result:

```
Python 3.1.2 (r312:79149, Mar 20 2010, 22:55:39) [MSC v.1500 64 bit
(AMD64)] on win32

Type "copyright", "credits" or "license()" for more information.

>>> ================================ RESTART
================================

>>>

total area of red glass: 23

>>>
```

These functions were typed into a new IDLE window. From the “File” drop-down menu of the IDLE menu bar select option “new” to open a second window. The example window shown below was named redGlass. All the code was typed there and execution was started by selecting the “run module” option of the “Run” menu.

Notice functions can return a result. Function red returns the total area of red glass. Our final line calls the print function with two parameters: the string literal “total area of red glass: “ and the results of the call to the red function. The print function concatenates and displays that literal with the result.

Saving your python code in a file allows you to continue to improve your functions and to build a portfolio of your work. When you want to continue working on that file you can use the open option of the file menu provided by IDLE. That open option allows you to search and find your saved file.

Suppose you have coded a collection of functions all of the same genre then you could save them in a separate file. For example the previously described square and rectangle functions could be stored in a file name myMath.py.

Your python code use the functions you have stored in file myMath.py. You type the import command. A problem might occur when your file is not in the path statement. To update that path command you import the stand os and sys modules. Then you append the new path to the path statement already in the sys module.

A module is simply a file containing functions written in python. Both the os and sys modules are automatically loaded when you install the python package on your computer, so you can safely import both of those modules.

Next you create a variable. The name lib_path is a convenient name. Any name would be acceptable for a new variable. To create an absolute path to your desktop you use the abspath method. That method is in the path module, so the command is path.abspath(). The command is always the module name followed by a dot and then the function name. The module path is with the module named os, so the entire command is os.path.abspath(). Notice the path is for a Windows machine. Normally the windows operating system uses back slashes for its path command, but our python functions follow the Unix custom of using forward slashes.

The append command attaches our new path statement to a list of such path statements. Later we still study how python handles lists.

Printing the path statement from the shows that the correct path has been added to the end of the IDLE's list of path statements. Now we can freely store our new files on the desktop. This hint came from stackoverflow.com/questions/279237/import-a-module-from-a-relative-path 8/29/2014

Important math functions have already been defined and stored in the module name math. To use those math functions, such as the sin function, you would begin your python file with the import math statement. The trigonometric functions work with angles measured in radians, not degrees, so use the command maxRadians = math.radians(maxAngle) to convert the angle in degrees to radians.

```
>>> print("180 degrees in radians: ", math.radians(180))

180 degrees in radians: 3.14159265359
```

Chemistry problem: Calculate the atomic weight of a molecule of propane, C_3H_8. Wikipedia provides basic information concerning propane at http://en.wikipedia.org/wiki/Propane. Write a function for the atomic weight of the hydrogen component. Write a separate function to return the atomic weight of the oxygen component Write a third function for the atomic weight of the carbon component. Find the chemical formula for propane. Calculate its atomic weight by calling the three functions for the atomic weight of carbon, hydrogen, and oxygen. Those atomic weights are available at http://en.wikipedia.org/wiki/Atomic_weights#Periodic_table_with_relative_atomic_masses.

The python code might be propane = carbon(3) + hydrogen(8).

Likewise you can calculate the right side of this chemistry transformation.

$$\mathbf{C_3H_8 + 5\ O_2 \rightarrow 3\ CO_2 + 4\ H_2O + heat}$$

propane + oxygen → carbon dioxide + water

The python code might be combustion = 3 * (carbon(3) + oxygen(2)) + 4 * (hydrogen(2) + oxygen(1)).

Another approach would be:

carbondioxide = 3 * (carbon(3) + oxygen(2))

water = 4 * (hydrogen(2) + oxygen(1))

combustion = carbondioxide + water

Engineering:

A two stage ladder has two segments each five meters in length. When extended the ladder segments must overlap by at least 20 %. When erected against a vertical wall the ladder must have a base angle between 70 degrees and 83 degrees. What is the maximum height this ladder can reach?

Hint: review you knowledge of trigonometry. Develop two functions. The first function decides the maximum length of the ladder. The second function calculates the maximum height when placed against a vertical wall.

Physics problem:

A golfer used three strokes to put the ball in the hole. The first stroke traveled 3.64 meters due North, the second went 2.92 meters at a 20 degree angle East of North, and the last rolled 1.23 meters due East. What would have been the distance and angle for a single shot to replace those three strokes? Hint: review your knowledge of trigonometry. Develop a function for calculating the next position. As parameters you will give the function the x and y coordinates of the current location, the angle for the next move, and the distance for the next move. Start at location (0,0).

Chapter 3: Getting started with Python Decisions

Purpose: Alter the flow of program execution

Python can respond based on conditions such as the values in parameters. This program gets an input parameter named **n**. That name was chosen because the expected input value would be an integer. The letters n and m are commonly used to designate integer values. If the parameter n contains a five then this program prints “good guess” else “try again.” Perhaps a better name for this function would be **guess_my_number**.

```
>>> def guess(n):

if n == 5.:

print("good guess")

else:

print("try again")


>>> guess(2)

try again

>>> guess(5)

good guess

>>>
```

Each time you call function **guess** you enter a number as a parameter. When you guess any number other than five the function responds with “try again.” Finally a guess of five produces a response of “good guess.” Notice the symbol for equality is a series of two equal symbols. Using only a single equal symbol would assign a value of five to variable **n**. Such a programming error might not produce an error message or warning. Thus programmers must carefully write and understand their programs. Thankfully the IDLE Python shell noticed the keyword **if** and produced this warning: “invalid syntax.”

Although not all python error messages are as cryptic as “invalid syntax” almost all python error messages can be solved by observing the highlighted command and rereading that command within its context in the program function.

A python function can accept a list of parameters. Consider someone searching for an apartment. Price, distance from work, and quality are some of the important qualities. Suppose each of those characteristics can be summarized as numbers. Price would be upwards from zero. Distance could be the number of miles or kilometers from your place of employment. And, quality might be scored on a scale from 1 to 5. Those three questions can be strung together by using the **and** operator. The **if** statement has three comparisons. Because those comparisons are connected by **and** operators only when all three comparisons return **true** will the **print(“possibility”)** statement be executed. If even one of those three comparisons returns **false** the **else** command will be executed.

```
def apt(price, distance, quality):
if price <= 400.00 and distance < 5 and quality >= 2:

print("possibility")

else:

print("out of the running")
```

Naturally our python program should query the user for each of those three ratings. Python 3 has the **input** command. For example **input('price')** will return the user's answer.

As shown in blue the **input** command displays the prompt of “What is the price?” The user's input of 2500 is shown in black. The **type** command shows that 2500 is an object of class string abbreviated as **str**.

All computations require numeric values, not strings, so the ask_price function uses the **int** function to convert that string value to an integer value. Notice how the following python program uses the **int** function to change the input value to an integer.

```
def apt(price, distance, quality):

if price <= 400 and distance < 5 and quality >= 2:

print("possibility")

else:

print("out of the running")
```

```
def ask_price():

price = input('What is the price? ')

return int(price)


def ask_distance():

distance = input('What is the distance? ')

return int(distance)


def ask_quality():

quality = input('What is the quality? ')

return int(quality)


def check_apt():

cost = ask_price()

far = ask_distance()

looks = ask_quality()

apt(cost, far, looks)

print('bye')


check_apt()
```

When run that python program can produce the following result.

```
>>>
What is the price? 250
What is the distance? 3
What is the quality? 2
possibility
bye
>>>
```

Python is an interpreted language. IDLE reads the file from the top. When reading a **def** command IDLE will store the function and all its commands for later execution. Each command within a function is indented once, so all the commands line up. The blank lines separating functions make the program easier for humans to read. Of course such extra blanks are meaningless to a computer program such as IDLE, so IDLE ignores blanks. When the next line begins at the left edge then the previous function has been completed. The commands always call functions that are above themselves, so IDLE already is aware of the meaning of the function name. When you select the run option IDLE begins at the top of the file storing functions and running commands. The entire file is a collection of functions to be stored for later execution and a single command on the last line. Many python programs conform to that program structure. Hence programmers often prefer to read their programs from the bottom to top while following the sequence offunction calls.

Calculating the price of a pizza. Suppose your local pizzeria makes three sizes of pizza: small for $5.99, medium for $6.99, and large for $7.99. The round pizzas are 8, 10, and 12 inches in diameter. The square pizzas are 7, 9, and 11 inches on edge. Stuffing the edges with extra cheese costs $1 more. All pizza comes with cheese and one free topping. A second topping costs 50 cents more. A third topping costs an additional 50 cents.

```
def pizza(size, shape, edge, extra1, extra2, extra3):
price = 5.99
if size == 'large':
price = 7.99
elif size == 'medium':
price = 6.99
else:
price = 5.99
if shape == 'round':
price = price + 0.50
if edge == 'stuffed':
price += 1
if extra2 != ' ':
price += 0.5
```

```
if extra3 != ' ':

price += 0.5

print("price is ", price)
```

```
pizza('medium', 'round', 'stuffed', 'cheese', 'sausage', 'pepperoni')
```

The program printed “price is 9.49.”

The first line is not needed because the next five lines cover all the options. In spite of its lack of need programmers often begin by setting the variable to a base value. Thus the programmer ensures that the variables always contains a valid value.

```
if size == 'large':

price = 7.99

elif size == 'medium':

price = 6.99

else:

price = 5.99
```

The **elif** operator combines an **else** operator with an **if** operator thus the program is shorter and easier to understand. The **!=** operator means **not equal**.

A source of error would occur if extra1 was set to blank while both extra2 and extra3 were chosen. Such an error would cause the customer to be overcharged by 50 cents. Such errors are not obvious and no warning would be given by IDLE. Because your design of the function always begin with using option 1, then option 2 and finally option 3 you might never think of using the function's parameters in a different manner. Testing of your function would have to be complete for you to catch that error. To avoid such problems programmers often begin by writing a list of conditions and the function's expected results. The function would be tested by using those hand-written cases.

Size large

shape round

edge stuffed

extra1 pepperoni

extra2 none

extra3 none

expected price is 7.99

Mathematics Problem:

Given the length of the three sides of a triangle your python program should check that the three sides can make a triangle. (Hint: when the sum of the lengths of two sides is less than the length of the longest side then those lengths cannot form a triangle.) Calculate its area and perimeter. Determine if the triangle is equilateral, isosceles, right, or obtuse.

Bioinformatics Problem:

Ask the user to type their age. Then calculate that person's remaining longevity. Sources of data are available at the Centers for Disease Control and Prevention Website at http://www.cdc.gov/nchs/fastats/life-expectancy.htm. Refine that estimate by using gender information.

Business Problem:

Your client can request a variety of nails. Straight steel nails with heads and length of three inches cost $1.10 per box of 100. Straight steel nails with heads and length of two inches cost $0.66 per box of 100. Straight steel nails with heads and length of one inch cost $0.43 per box of 100. Screw nails cost an additional ten cents per box. Galvanization costs an additional twenty-three cents per box. Straight copper nails with heads cost $3.42 per box. Copper nails are not available as screws, nor is galvanization allowed.

Engineering Problem:

Gaming Problem:

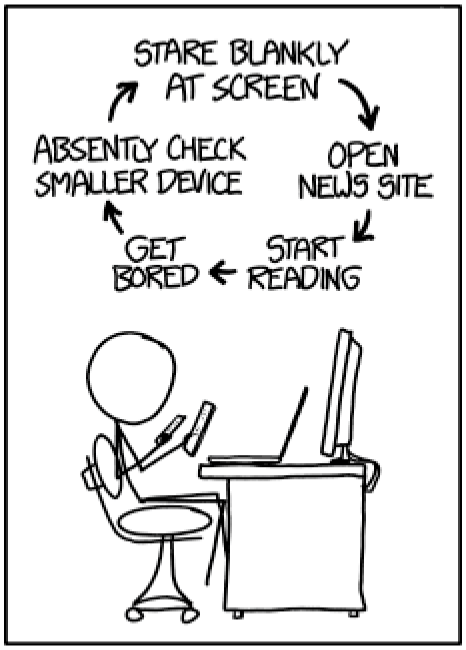
Design a maze of a small number of rooms each containing optional treasure, enemy, and most importantly doors. The player starts with $5. For each room, the player defeats or loses to the room's enemy depending on the amount of money it costs. Next the players gathers up the valuables which are represented by a dollar amount. Finally the player selects which door from a list of one to three doors. To survive all the rooms is to succeed. To die is to lose the game. There are no visuals in this game. Everything is text.

Topic: Recursion

Functions can call other functions and even call themselves. Recursion involves a function repeatedly calling itself until the goal is satisfied.

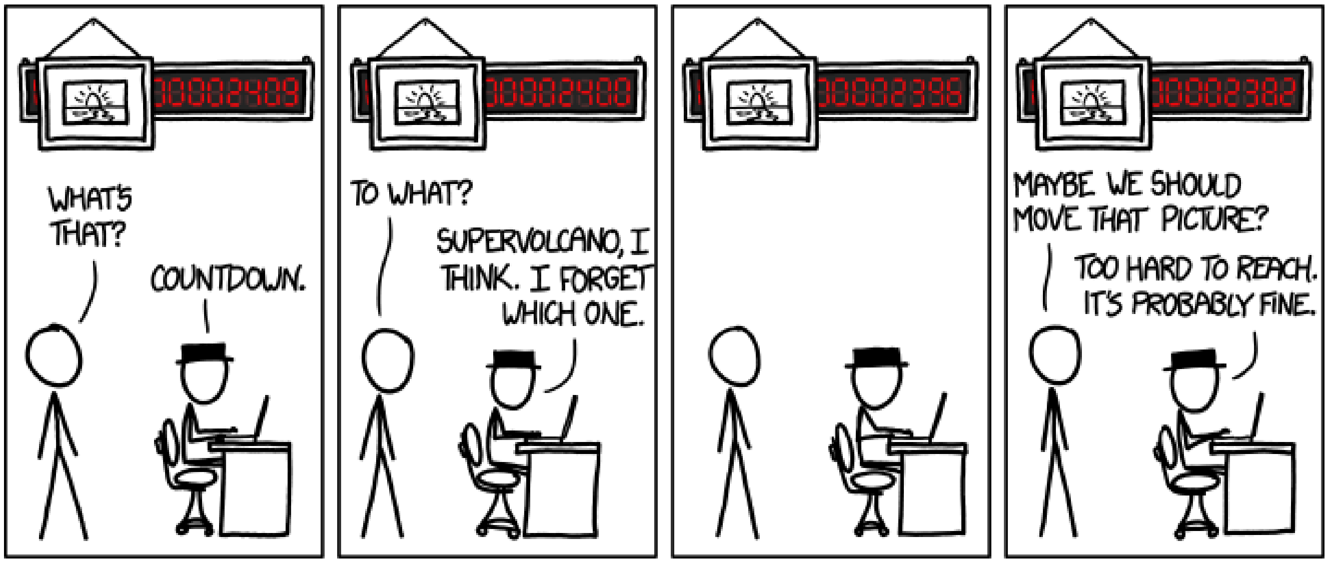
Chapter 4: Getting started with Python Recursion

Topic: Loop



http://xkcd.com/1411/

A loop involves repeatedly executing the same sequence of instructions. While a loop without an end is useful for some permanently installed devices, most application programs require a check for an end condition.


00002409
WHAT'S THAT?
COUNTDOWN.
00002400
TO WHAT?
SUPERVOLCANO, I THINK. I FORGET WHICH ONE.
00002396
00002382
MAYBE WE SHOULD MOVE THAT PICTURE?
TOO HARD TO REACH. IT'S PROBABLY FINE.

http://xkcd.com/1159/

Topic: Recursion

Functions can call other functions and even call themselves. Recursion involves a function repeatedly calling itself until the goal is satisfied. A simple example is a countdown problem. Some would count down to a rocket's launch. Others would countdown to a volcano's eruption. The function to be repeated called should begin by checking for completion of its task.

Running the countdown function printed this result.

Almost all the examples of recursion place the end termination question immediately at the top of the function. The authors of those example programs are trying to encourage a safe style of programming. Any mistake with that termination question or the change of the current counter could lead to an infinite loop. Thankfully the IDLE monitor will automatically halt execution of an infinite loop, but only after a large number of iterations.

Students of recursion often cannot describe what happened during the execution of a recursive program. All they know is that the countdown did print from 5 down to 2. How might you change that program to print from 5 down to 1? One clue would be to examine the termination question “start <= end”. When the count is down to 1, then start is one and end is of course always 1, so the function terminates with a return statement. The function does not print “1”. Change the termination question to “start < end”.

Now the function will print “1”.

Obviously we should carefully study the execution of this program. One method is to draw an execution tree. Each box on the left side has the call with the current parameters which vary from 5 down to 0. The box on the right side has the displayed text. The arrows with solid lines show the flow of execution from countdown(5, 1) down to countdown(0, 1). The dashed arrows display the connection from the execution instance to the corresponding potion of the display. The countdown(0, 1) instance does not display a 0, instead countdown(0, 1) executes the return command. Thus there is no dashed line from countdown(0, 1).

Topic: Fibonacci numbers: How many rabbits? One mathematician's estimate

http://en.wikipedia.org/wiki/Fibonacci_number describes Fibonacci's estimate for the number of rabbits that a single pair can produce. Being a mathematician he set up unrealistic criteria: rabbits never die, a pair of rabbits always produce another pair of rabbits with one being male and the other female on schedule every month after the second month.

The first few Fibonacci numbers are:

Fib(0) = 0

Fib(1) = 1

Fib(2) = 1

Fib(3) = 2

Fib(4) = 3

Fib(5) = 5

Fib(6) = 8

Fib(7) = 13

Fib(8) = 21

Fib(9) = 34

Fib(10) = 55

Where the general rule is Fib(n) = Fib(n-1) + Fib(n-2) with the assumption that the series begins with Fib(0) = 0 and Fib(1) = 1.

Our python python program follows that rule. For fib(0) and fib(1) our program returns 0 and 1 respectively.

For the fifth Fibonacci number fib(5) the program correctly printed 5. Although repeatedly testing can quickly convince us of the correctness of this program, an understanding of the sequence of function calls remains important. To help us see the series of call we insert a print statement with an offset of three dashes (---) incremented by an additional three dashes each time the recursive function is called.

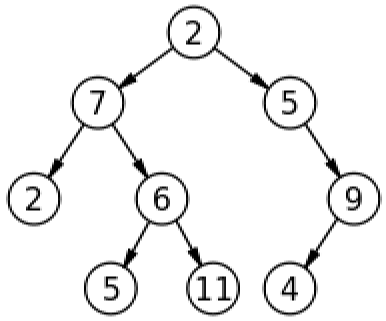
This program's execution produced the following listing.

Notice how each method call immediately sets up the next method call until the base cases of fib(0) or fib(1) are encountered. This sequence of method calls sets up a stack of method calls: fib(5), fib(4), fib(3), fib(2), and finally fib(1). As the bottom method call is solved and the value is returned to the immediately preceding calling method. Slowly the stack is completelyt undone and the result is printed.

Bio-informatics Problem:

http://en.wikipedia.org/wiki/Basic_reproduction_number describes the basic reproductive ratio, R0, of an infection as the average number of new cases generated by each current case of the disease. Assume an infectious period of three weeks and a death rate of 50%. Beginning with one sick patient, how many would become infected and how many deaths would occur over a period of 10 cycles of 3 weeks each?

Computer Science



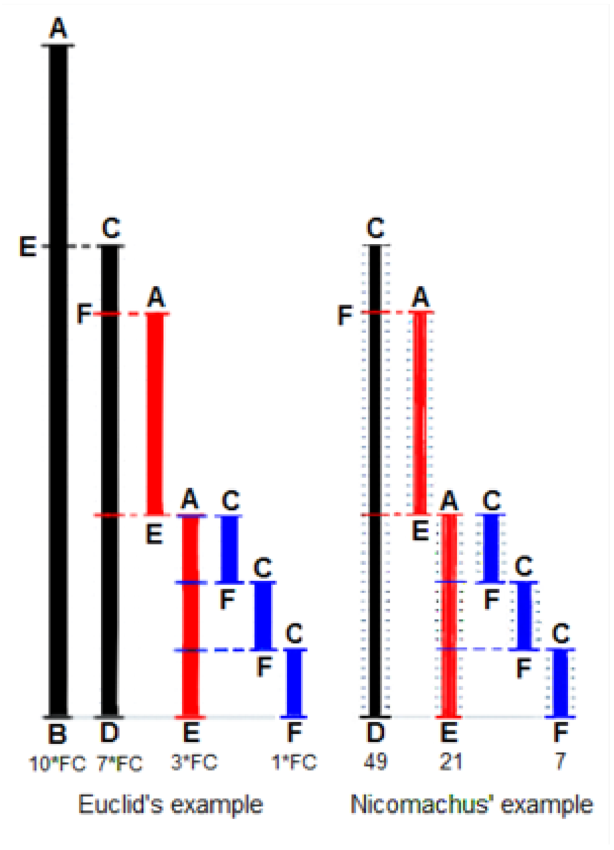
In a binary tree, which begins at the top, each node can hold one piece of information and connect to at most two child nodes displayed beneath it. How many nodes can a tree hold when the depth, meaning the number of levels of nodes, is 10.

Economics problem

http://en.wikipedia.org/wiki/Chain_reaction describes chain reactions including a section Chain Reactions in Economics. Assuming the collapse of one bank will cause the collapse of two more banks within the next month, how many banks would collapse in a year?

Mathematics problem

http://en.wikipedia.org/wiki/Euclidean_algorithm describes the Euclidean algorithm.



Euclid's method for finding the greatest common divisor (GCD) of two starting lengths BA and DC, both defined to be multiples of a common "unit" length. The length DC being shorter, it is used to "measure" BA, but only once because remainder EA is less than CD. EA now measures (twice) the shorter length DC, with remainder FC shorter than EA. Then FC measures (three times) length EA. Because there is no remainder, the process ends with FC being the GCD. On the right Nicomachus' example with numbers 49 and 21 resulting in their GCD of 7 (derived from Heath 1908:300).

http://en.wikipedia.org/wiki/Recursion_(computer_science) describes how the Euclidean algorithm can be written in a recursive format.

Function definition*:*

$$\gcd(x,y) = \begin{cases} x & \text{if } y = 0 \\ \gcd(y, \operatorname{remainder}(x,y)) & \text{if } y > 0 \end{cases}$$

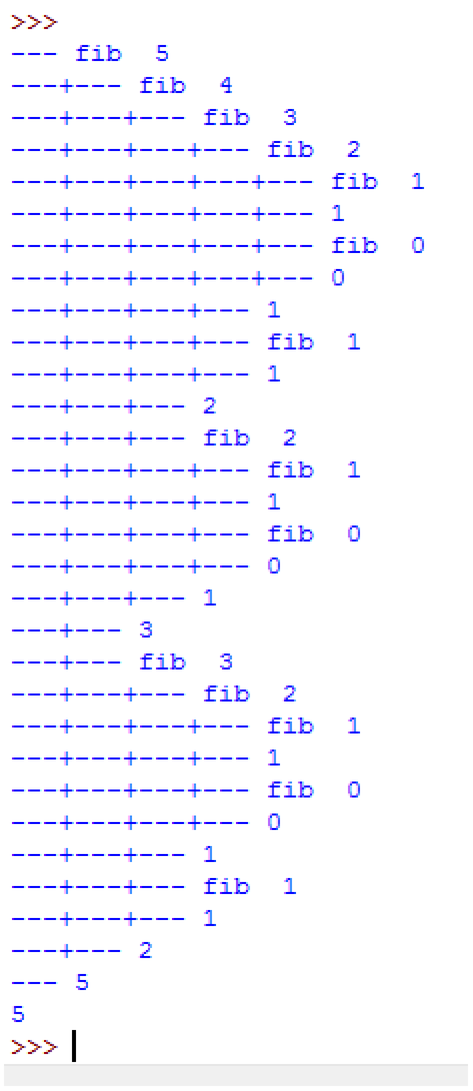
Implement that recursive form of the Euclidean algorithm in python.

Packaging Science problem:

Your idea for tightly packaging a cube of dimensions 4.8 units by 4.8 units by 4.8 units with four cubes each of dimensions 2.3 units by 2.3 units by 2.3 units and then packing each of those smaller cubes with four even smaller cubes of size 1.1 units by 1.1 units by 1.1 units. The packaging has a thickness of .1, so these cubes fit tightly inside each other with a gap of 0.1. How many boxes are used? What is the total volume of packaging? What is the total volume of useful space?

Physics problem:

http://en.wikipedia.org/wiki/Chain_reaction describes a chained reaction such as occurs in an atomic bomb. Assuming the explosion begins with a single nuclear breakdown that causes a cascading of two more such events every nanosecond. How many nuclear events would occur after 10 nanoseconds? After 100 nanoseconds?



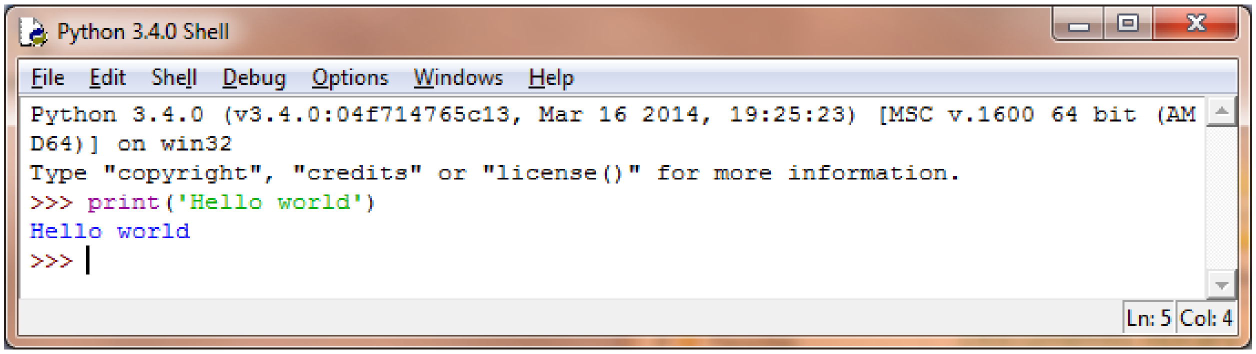
```
>>>
--- fib  5
---+--- fib  4
---+---+--- fib  3
---+---+---+--- fib  2
---+---+---+---+--- fib  1
---+---+---+---+--- 1
---+---+---+---+--- fib  0
---+---+---+---+--- 0
---+---+---+--- 1
---+---+---+--- fib  1
---+---+---+--- 1
---+---+--- 2
---+---+--- fib  2
---+---+---+--- fib  1
---+---+---+--- 1
---+---+---+--- fib  0
---+---+---+--- 0
---+---+--- 1
---+--- 3
---+--- fib  3
---+---+--- fib  2
---+---+---+--- fib  1
---+---+---+--- 1
---+---+---+--- fib  0
---+---+---+--- 0
---+---+--- 1
---+---+--- fib  1
---+---+--- 1
---+--- 2
--- 5
5
>>>
```

You should trace that execution of our python program. The indentation using dashes and a + symbol reflect the depth of the stack of calls to this function. This function immediately produces a stack of calls from fib(5) down to fib(1) before any returns happen. By lining up the function call with the return you can understand when the matching return statement is executed.

Chapter 5: Strings

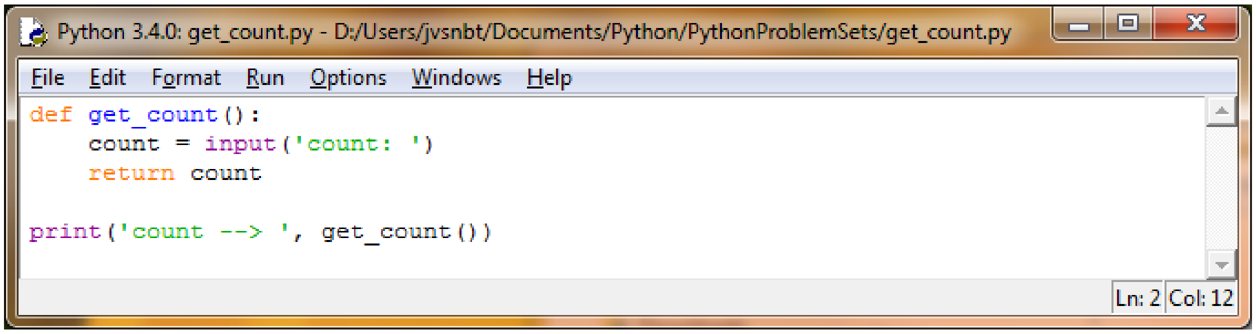
Topic: Strings

Previous python programs have displayed strings. For example the hello_world method displayed “Hello world.”



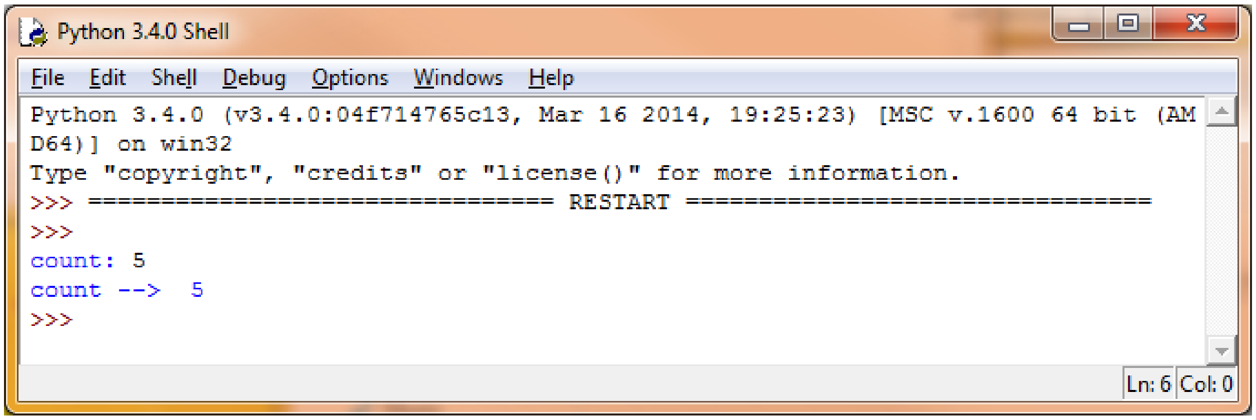
```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:25:23) [MSC v.1600 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print('Hello world')
Hello world
>>> 
```

Also the python method named input can use a string as a prompt.


Python 3.4.0: get_count.py - D:/Users/jvsnbt/Documents/Python/PythonProblemSets/get_count.py
File Edit Format Run Options Windows Help
Ln: 2 Col: 12

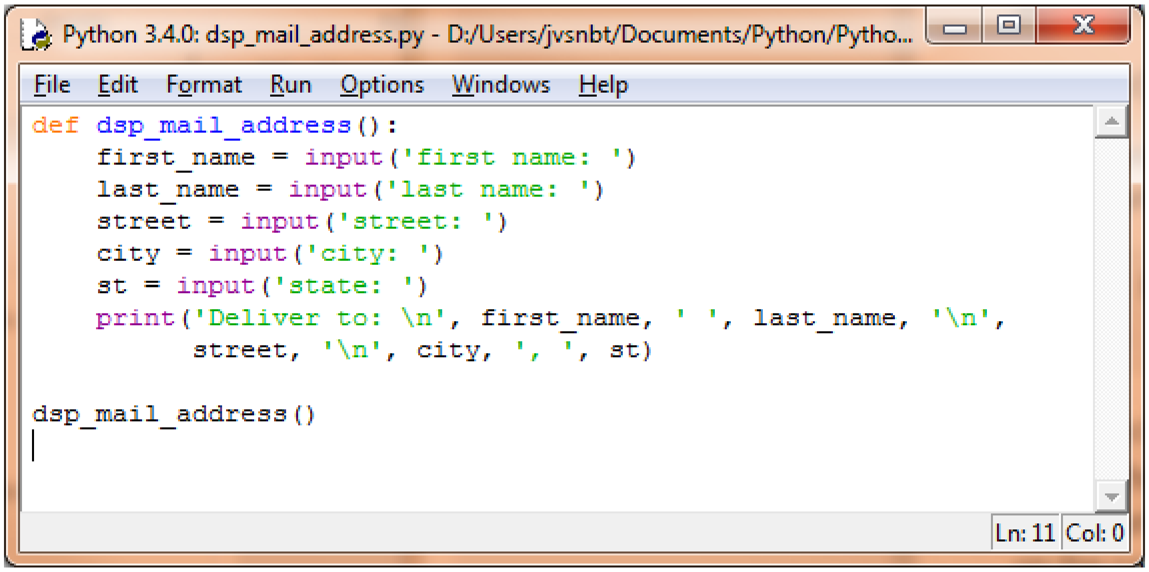
```
def get_count():
    count = input('count: ')
    return count

print('count --> ', get_count())
```

Notice also that the print command can display a list of objects separated by commas. Strings are often displayed as labels for values.



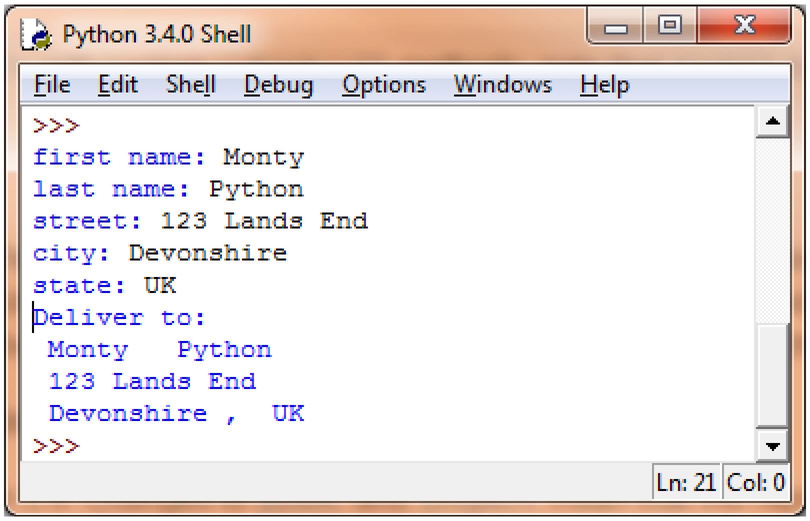
```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:25:23) [MSC v.1600 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
count: 5
count -->  5
>>>
```

The python print command also includes a newline character at the end of a printed line. The newline character is explained by Wikipedia at http://en.wikipedia.org/wiki/Newline. Python, like most computer languages, represents the newline character as '\n'. One python print command can print several lines by including newline characters.


Python 3.4.0: dsp_mail_address.py - D:/Users/jvsnbt/Documents/Python/Pytho...
File Edit Format Run Options Windows Help
Ln: 11 Col: 0

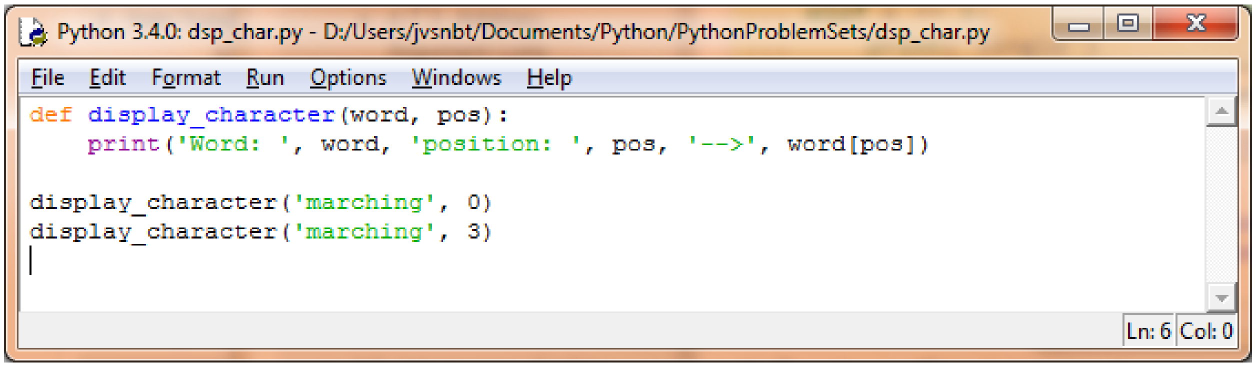
```
def dsp_mail_address():
    first_name = input('first name: ')
    last_name = input('last name: ')
    street = input('street: ')
    city = input('city: ')
    st = input('state: ')
    print('Deliver to: \n', first_name, ' ', last_name, '\n',
          street, '\n', city, ', ', st)

dsp_mail_address()
```

That module produced the following result.



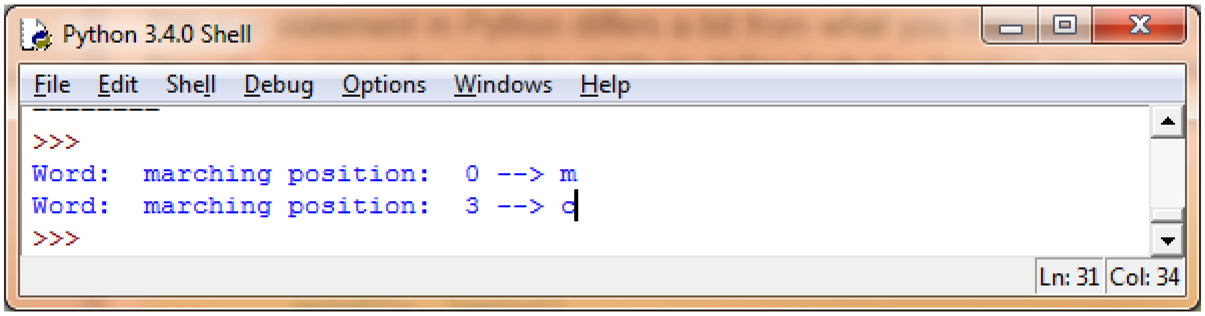
```
>>>
first name: Monty
last name: Python
street: 123 Lands End
city: Devonshire
state: UK
Deliver to:
 Monty    Python
 123 Lands End
 Devonshire ,  UK
>>>
```

Python can do much more with strings than simply printing them. A string is a list of characters, so a python method can loop through that sequence processing each character one at a time. That list of characters is indexed beginning from zero. This python method will display the character at the selected position within the string.


Python 3.4.0: dsp_char.py - D:/Users/jvsnbt/Documents/Python/PythonProblemSets/dsp_char.py
File Edit Format Run Options Windows Help
Ln: 6 Col: 0

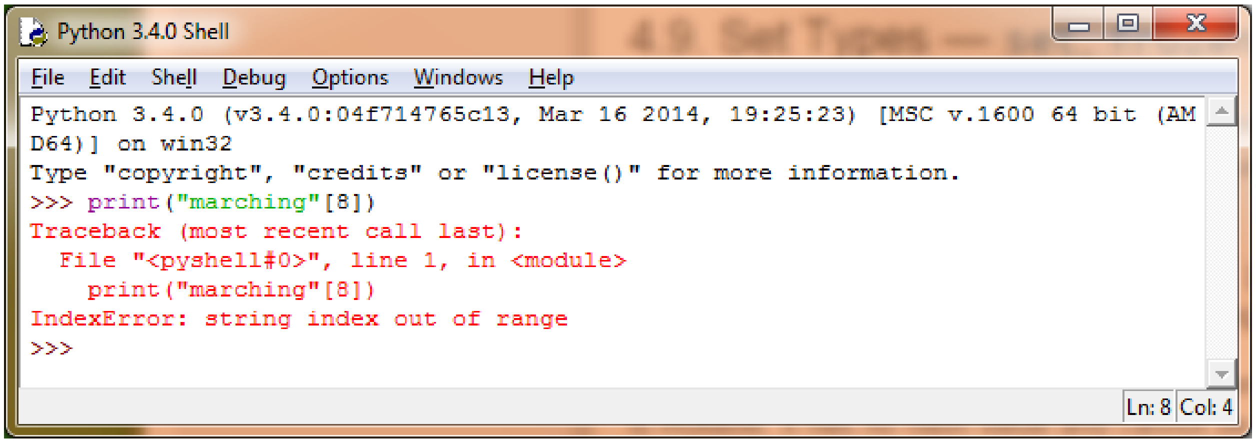
```
def display_character(word, pos):
    print('Word: ', word, 'position: ', pos, '-->', word[pos])

display_character('marching', 0)
display_character('marching', 3)
```

Displaying the character at position 0 and then position 3 produced this result.



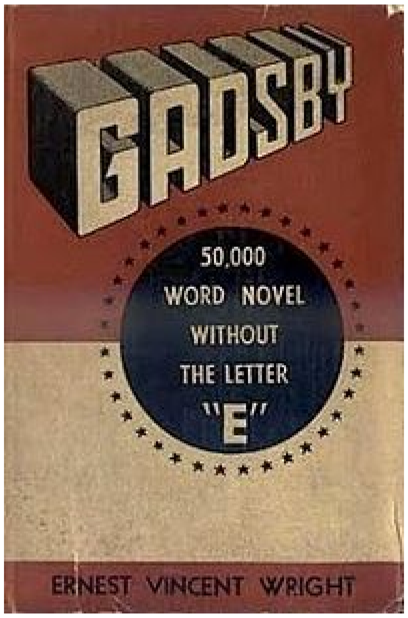
```
>>>
Word:  marching position:  0 --> m
Word:  marching position:  3 --> c
>>>
```

The word “marching” has a length of 8, so the indices vary from zero to seven. Asking for the 8th position will produce an error message.

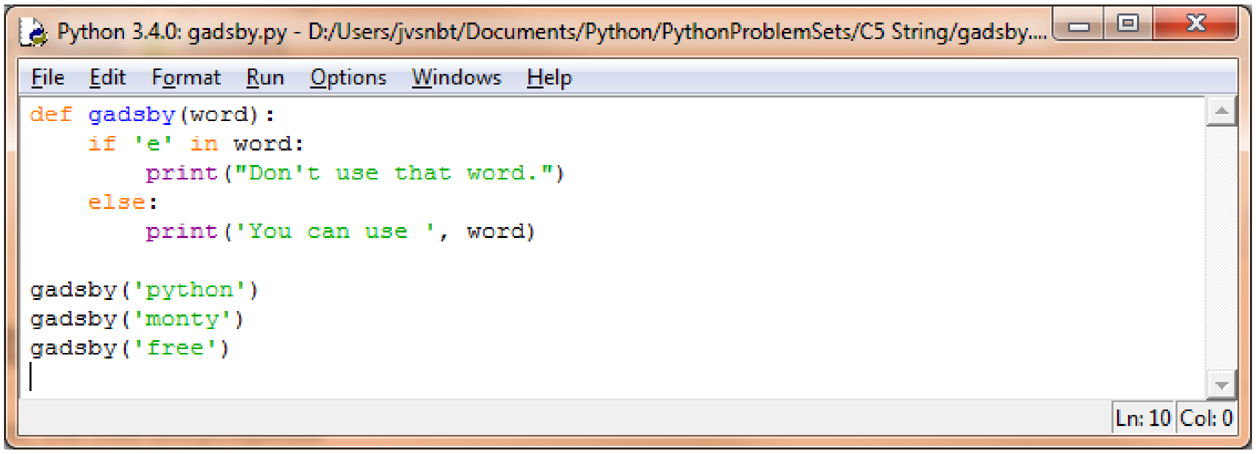


```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:25:23) [MSC v.1600 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("marching"[8])
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print("marching"[8])
IndexError: string index out of range
>>>
```

http://en.wikipedia.org/wiki/Gadsby_(novel)) describes how Ernest Vincent Wrights avoided the use of any word containing “e” to write Gadsby.

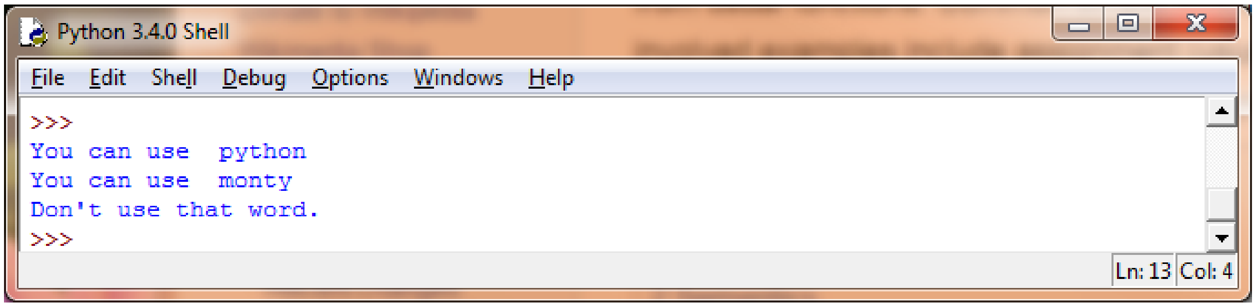


The operators used in the prior example python functions have been close to mathematics. Those operators included <, >, ==, +, -, and /. A special string operator is “in.”


Python 3.4.0: gadsby.py - D:/Users/jvsnbt/Documents/Python/PythonProblemSets/C5 String/gadsby....
File Edit Format Run Options Windows Help
Ln: 10 Col: 0

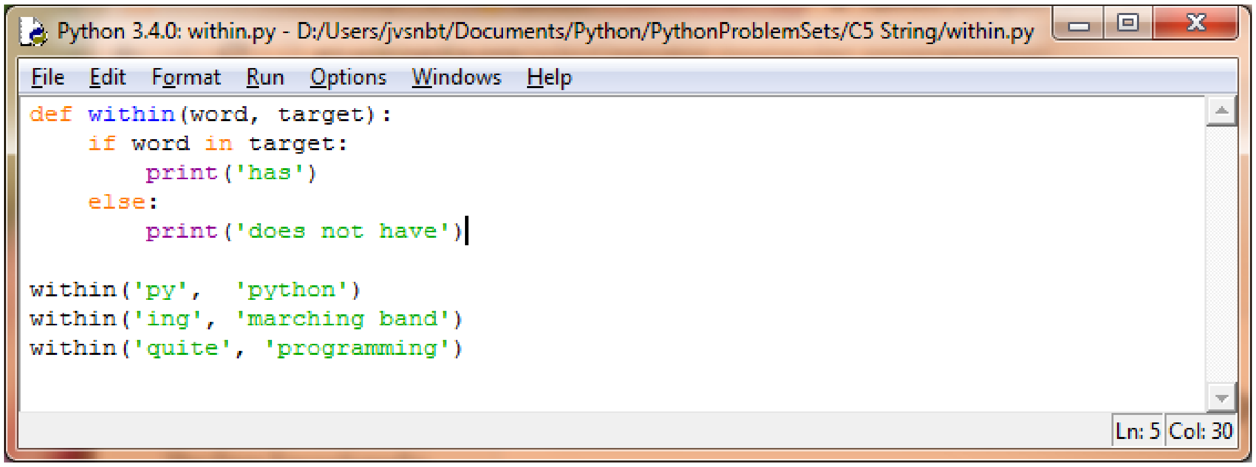
```
def gadsby(word):
    if 'e' in word:
        print("Don't use that word.")
    else:
        print('You can use ', word)

gadsby('python')
gadsby('monty')
gadsby('free')
```

As expected Gadsby could include “python” and “monty”, but not “free.” Notice that the word “don't” includes an apostrophe, so that string begins and ends with double quotation marks. Strings can begin and end with either double or single quotation marks. Thus a single quotation mark can be displayed in a message. Be careful that you begin and end the string with the same choice of double or single quotation marks. Don't begin with a single quotation mark and end the string with a double quotation mark.



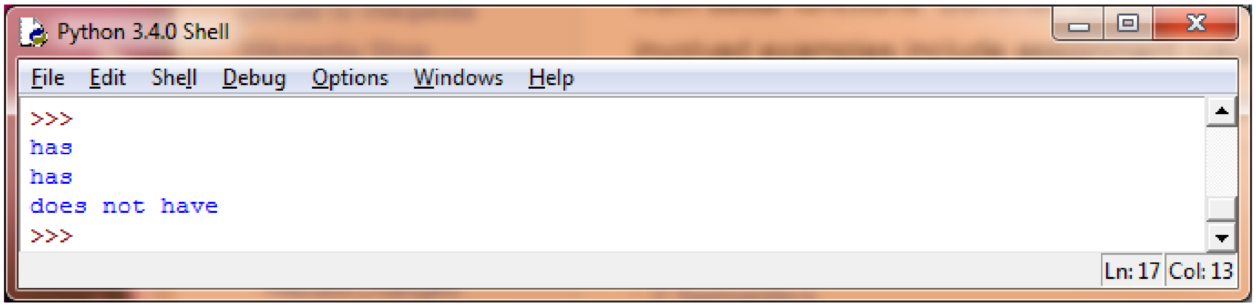
```
>>> 
You can use  python
You can use  monty
Don't use that word.
>>> 
```

Python's “in” operator will also check for an entire string being contained within another string.


Python 3.4.0: within.py - D:/Users/jvsnbt/Documents/Python/PythonProblemSets/C5 String/within.py
File Edit Format Run Options Windows Help
Ln: 5 Col: 30

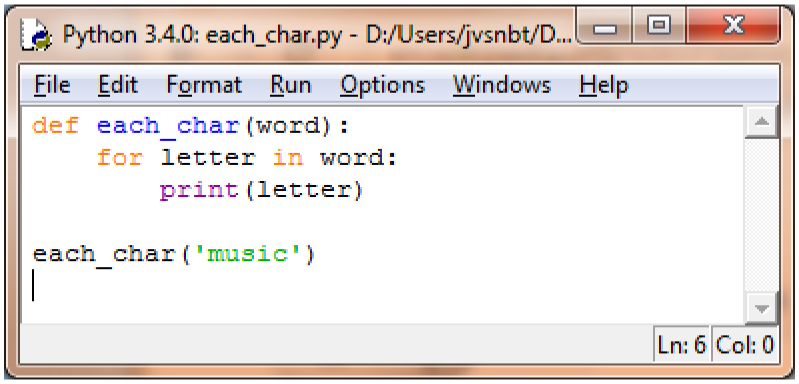
```
def within(word, target):
    if word in target:
        print('has')
    else:
        print('does not have')

within('py',  'python')
within('ing', 'marching band')
within('quite', 'programming')
```

As expected “python” contains “py.” Also “marching band” has “ing.” But, “programming” does not contain “quite.”



```
>>>
has
has
does not have
>>>
```

You can write a for statement the lists each character in a string.


Python 3.4.0: each_char.py - D:/Users/jvsnbt/D...
File Edit Format Run Options Windows Help
Ln: 6 Col: 0

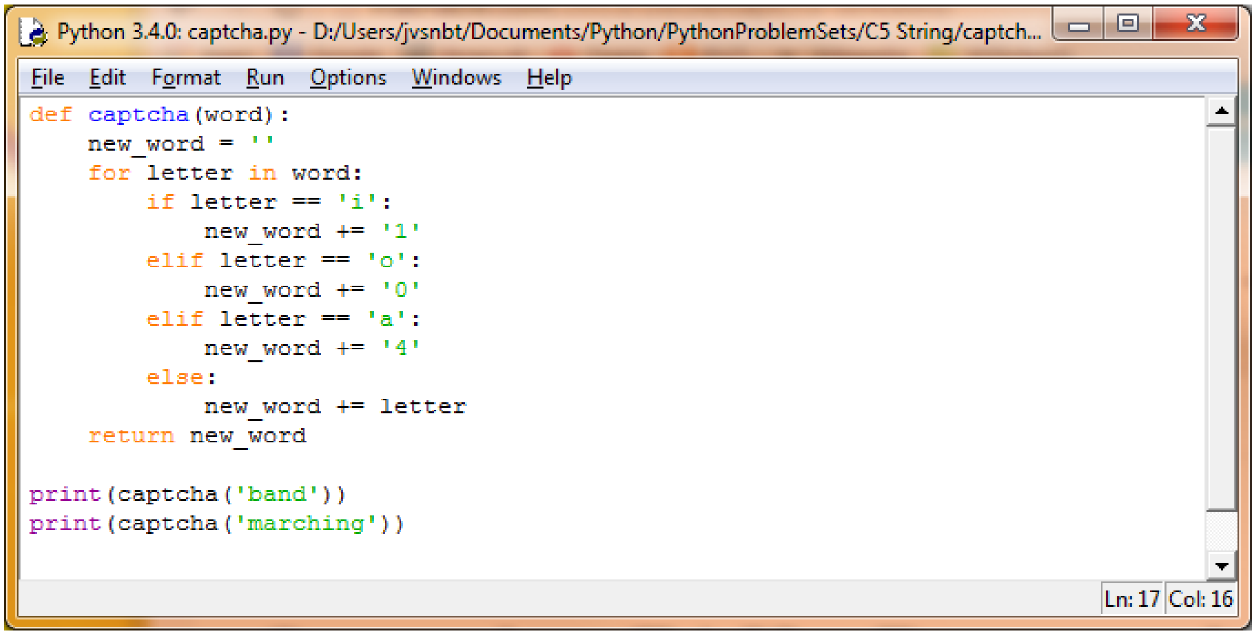
```
def each_char(word):
    for letter in word:
        print(letter)

each_char('music')
```

This function sets up a loop that prints each letter in the word beginning with the first letter, also known as word[0], to the last letter, also represented as word[3].

http://en.wikipedia.org/wiki/CAPTCHA

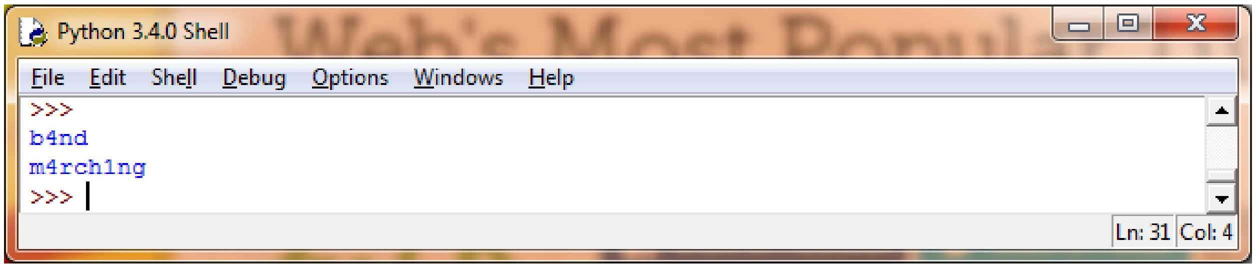
A captcha, as shown above and described at http://en.wikipedia.org/wiki/CAPTCHA, include distorted letters to prevent artificial intelligence (AI) programs from entering a Web site. Also a captcha might include a number or special character that is similar in shape to the replaced letter, thus further confounding any AI program. Perhaps the developer of the previous captcha thought "w" was similar enough to "u" to confound any AI program.

In our captcha function each letter "i" will be replaced with a number "1." Letter "o" will be replaced with the digit "0." And, "a" becomes "4."


Python 3.4.0: captcha.py - D:/Users/jvsnbt/Documents/Python/PythonProblemSets/C5 String/captch...
File Edit Format Run Options Windows Help
Ln: 17 Col: 16

```
def captcha(word):
    new_word = ''
    for letter in word:
        if letter == 'i':
            new_word += '1'
        elif letter == 'o':
            new_word += '0'
        elif letter == 'a':
            new_word += '4'
        else:
            new_word += letter
    return new_word

print(captcha('band'))
print(captcha('marching'))
```

Thus "band" becomes "b4nd" and "marching" is changed to "m4rch1ng."



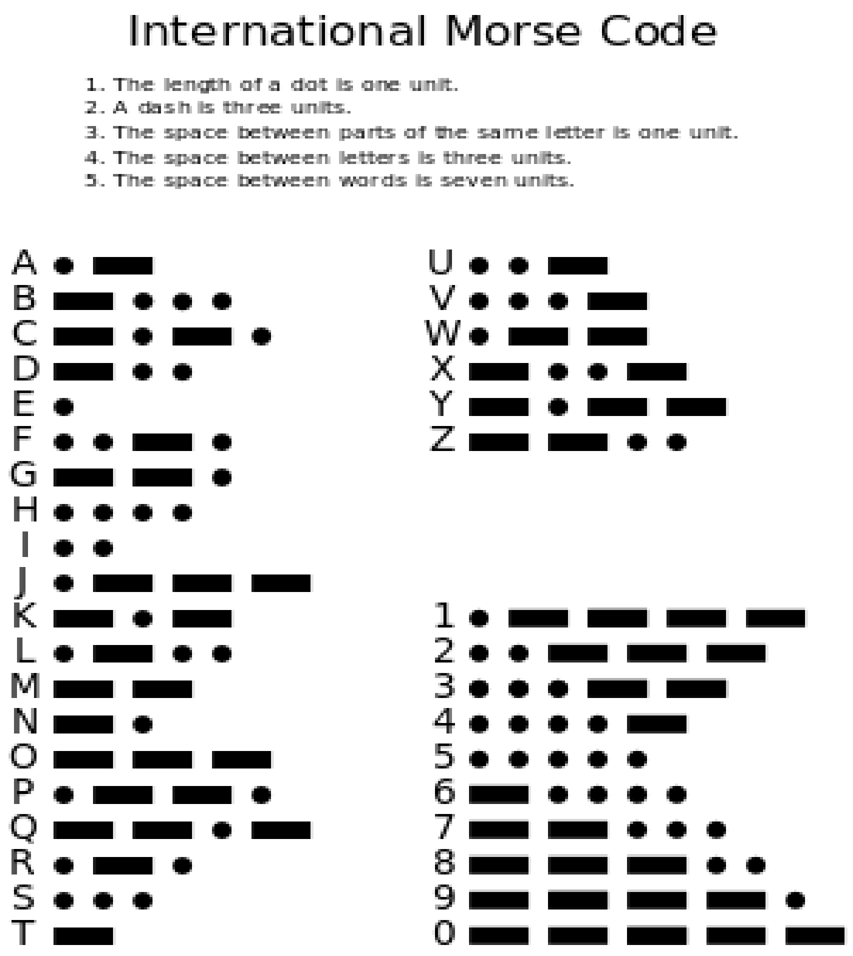
```
>>>
b4nd
m4rch1ng
>>> 
```

Although still commonly used to protect Web sites, computer security researchers have reported the defeat of captchas by AI programs at http://mashable.com/2013/10/28/captcha-defeated/.

Python provides an extensive list of predefined functions for use with strings, some of which could replace the above function, but those predefined functions require knowledge of more advanced topics within the python programming language.

Exercises:

Fun for all majors:



International Morse Code

1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.

A	● ▬	U	● ● ▬
B	▬ ● ● ●	V	● ● ● ▬
C	▬ ● ▬ ●	W	● ▬ ▬
D	▬ ● ●	X	▬ ● ● ▬
E	●	Y	▬ ● ▬ ▬
F	● ● ▬ ●	Z	▬ ▬ ● ●
G	▬ ▬ ●		
H	● ● ● ●		
I	● ●		
J	● ▬ ▬ ▬		
K	▬ ● ▬	1	● ▬ ▬ ▬ ▬
L	● ▬ ● ●	2	● ● ▬ ▬ ▬
M	▬ ▬	3	● ● ● ▬ ▬
N	▬ ●	4	● ● ● ● ▬
O	▬ ▬ ▬	5	● ● ● ● ●
P	● ▬ ▬ ●	6	▬ ● ● ● ●
Q	▬ ▬ ● ▬	7	▬ ▬ ● ● ●
R	● ▬ ●	8	▬ ▬ ▬ ● ●
S	● ● ●	9	▬ ▬ ▬ ▬ ●
T	▬	0	▬ ▬ ▬ ▬ ▬

Convert a string to Morse code. Wikipedia explain Morse code at http://en.wikipedia.org/wiki/Morse_code.

Computer Science:

Count the occurrences of a letter within a string. This is a basic starting strategy for code cracking. The frequency of the letters, such as “e,” depends on the language.

Business:

Software Engineering:

Software engineering projects often consist of massive quantities of modules, or functions, grouped that the core topic. A financial system might contain a billing module within which all functions begin with a capital letter “B.” Write a python function to check that the first letter of the module name is “B.”

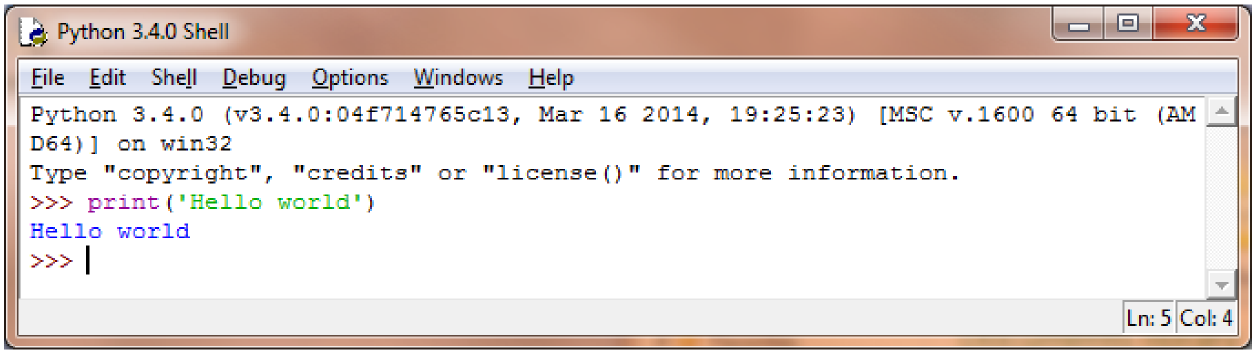
Biotechnology:

DNA, described by Wikipedia at http://en.wikipedia.org/wiki/DNA, consists of a series of letters. Write a python function called chk_DNA_letters to check that a string contains only those letters. Return either a boolean value of true or false.

Chapter 6: Sequences

Topic: Sequences

Previous python programs have displayed strings. For example the hello_world method displayed “Hello world.”



```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:25:23) [MSC v.1600 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print('Hello world')
Hello world
>>> 
```

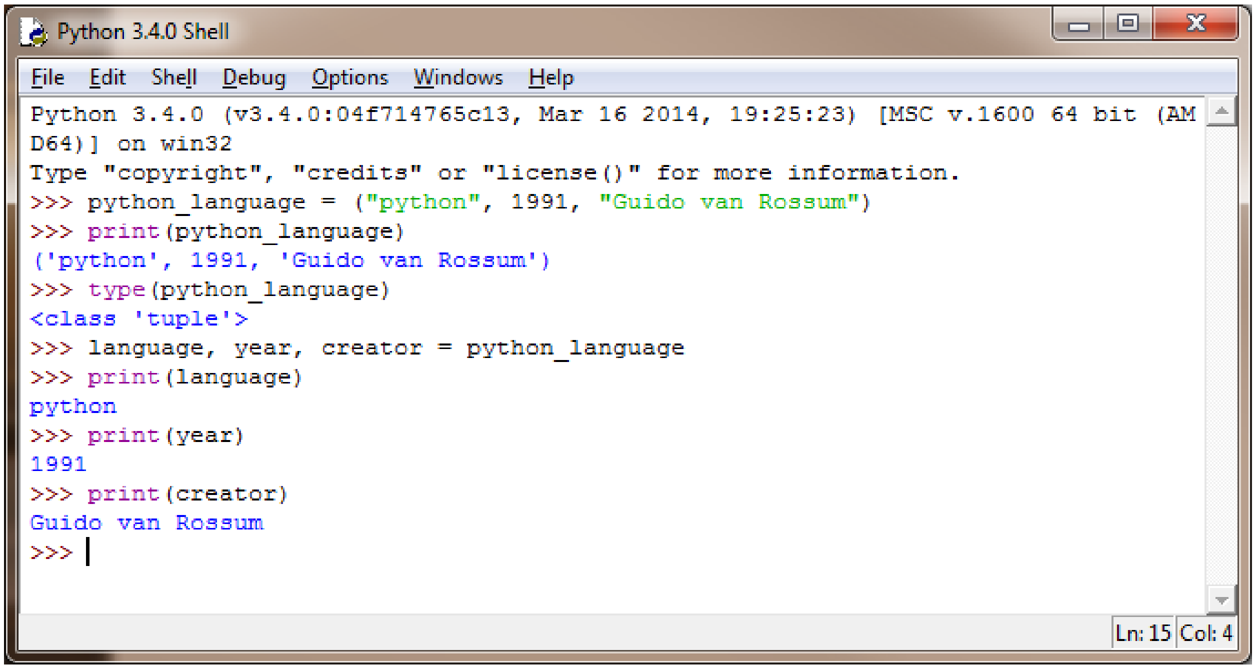
History: Hello, world!

Displaying “Hello, world!” has been a tradition within the arcane field of computer science texts. Wikipedia at http://en.wikipedia.org/wiki/%22Hello,_world!%22_program hints at its origin during, or prior to, 1974.

Python currently has three examples of sequences: strings, tuples, and lists. All of those sequences have common properties such as indexing. As an example the first character of the string “hello” is hello[0] which is 'h'. The **len** function returns the length of the sequence. The length of the string “hello” is 5, so len(“hello”) returns an **int** value of 5. The last character of the string “hello” is “hello”[4], which is the character 'o.'

Topic: Tuples

Python currently has three examples of sequences: strings, tuples, and lists. A tuple is an unchangeable, also known as immutable, sequence of python values and objects, separated by commas, and surrounded by parenthesis. An example tuple is (“python”, 1991, “Guido van Rossum”) which contains the year of python's first release and the creator of the python programming language.

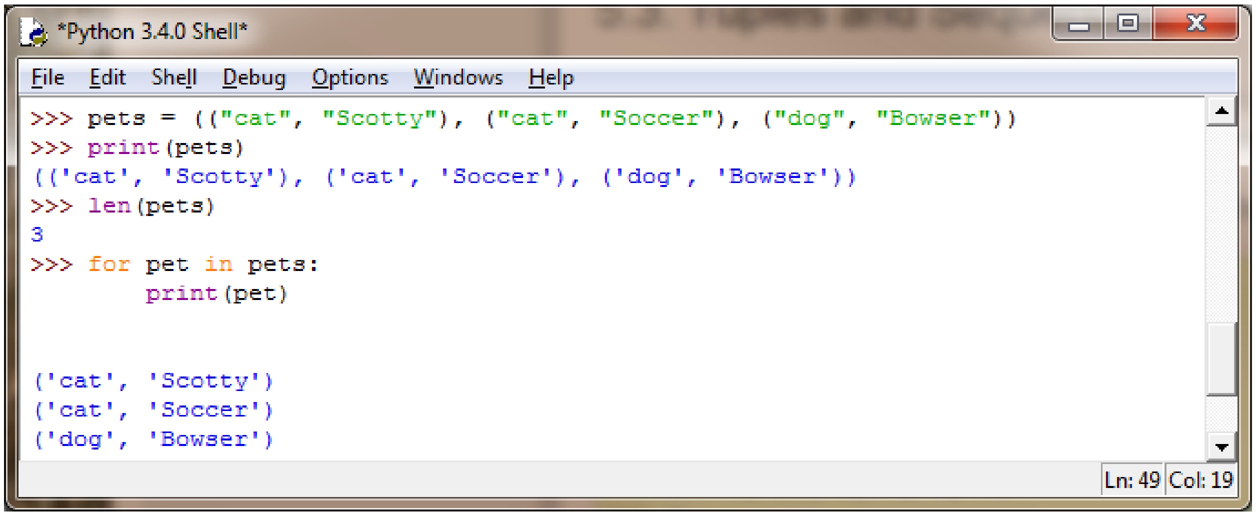


```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:25:23) [MSC v.1600 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> python_language = ("python", 1991, "Guido van Rossum")
>>> print(python_language)
('python', 1991, 'Guido van Rossum')
>>> type(python_language)
<class 'tuple'>
>>> language, year, creator = python_language
>>> print(language)
python
>>> print(year)
1991
>>> print(creator)
Guido van Rossum
>>> |
```

In this example the variable named python_language holds a tuple shown as (“python”, 1991, “Guido van Rossum”). The surrounding parenthesis shows this to be a tuple. Commas separate the items in that tuple. The print command displays that tuple as a sequence beginning and ending with a parenthesis. The type operator displays the type of the object which is a tuple.

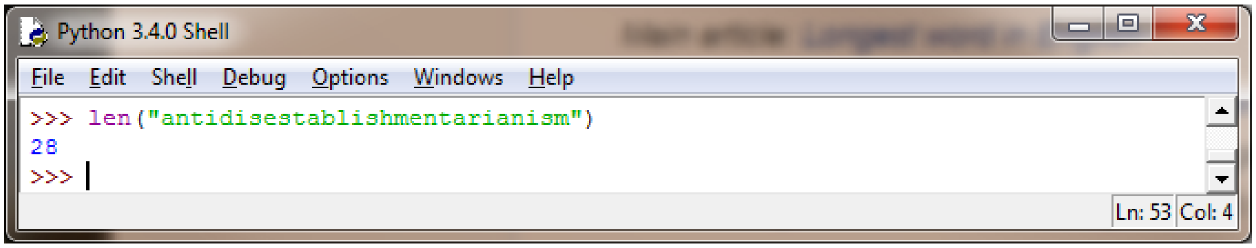
Python programs use a sequence of variables to extract the items from a tuple. For example in one statement the three variables language, year, and creator get values from the same tuple. Obviously the command must have as many variables are the tuple has items.

Suppose you had three pets: a cat named Scotty, a second cat named Soccer, and a dog named Bowser, then your python program could have a separate tuple for each pet. For Scotty the tuple would be (“cat”, “Scotty”). For Soccer the associated tuple would be (“cat”, “Soccer”). And for the dog Bowser the tuple would be (“dog”, “Bowser”). All three tuples could be combined into another tuple for all your pets.



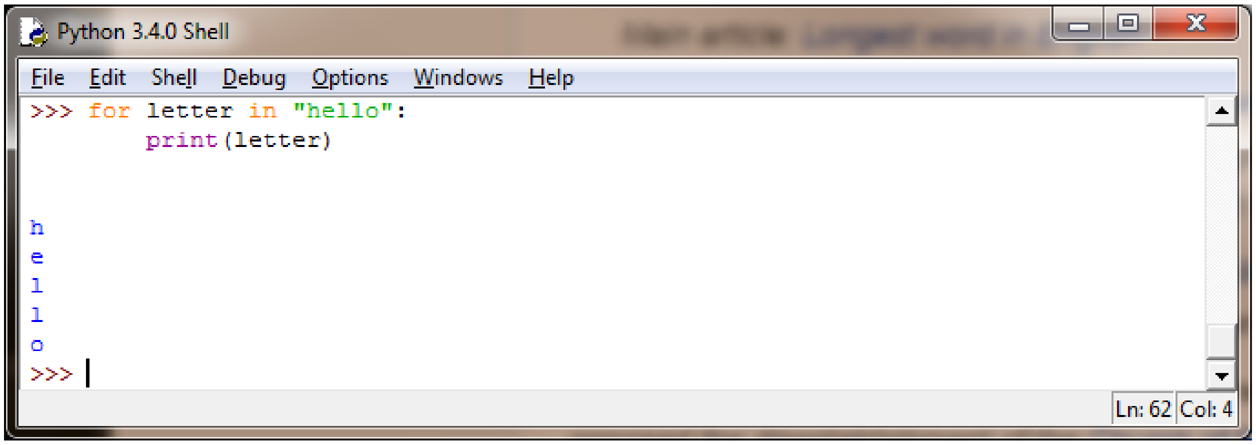
```
>>> pets = (("cat", "Scotty"), ("cat", "Soccer"), ("dog", "Bowser"))
>>> print(pets)
(('cat', 'Scotty'), ('cat', 'Soccer'), ('dog', 'Bowser'))
>>> len(pets)
3
>>> for pet in pets:
        print(pet)


('cat', 'Scotty')
('cat', 'Soccer')
('dog', 'Bowser')
```

The len function returned the length of a string. Thus len(“antidisestablishmentarianism”) is 28. Strings are examples of sequences and the len function works with any sequence.



```
>>> len("antidisestablishmentarianism")
28
>>> 
```

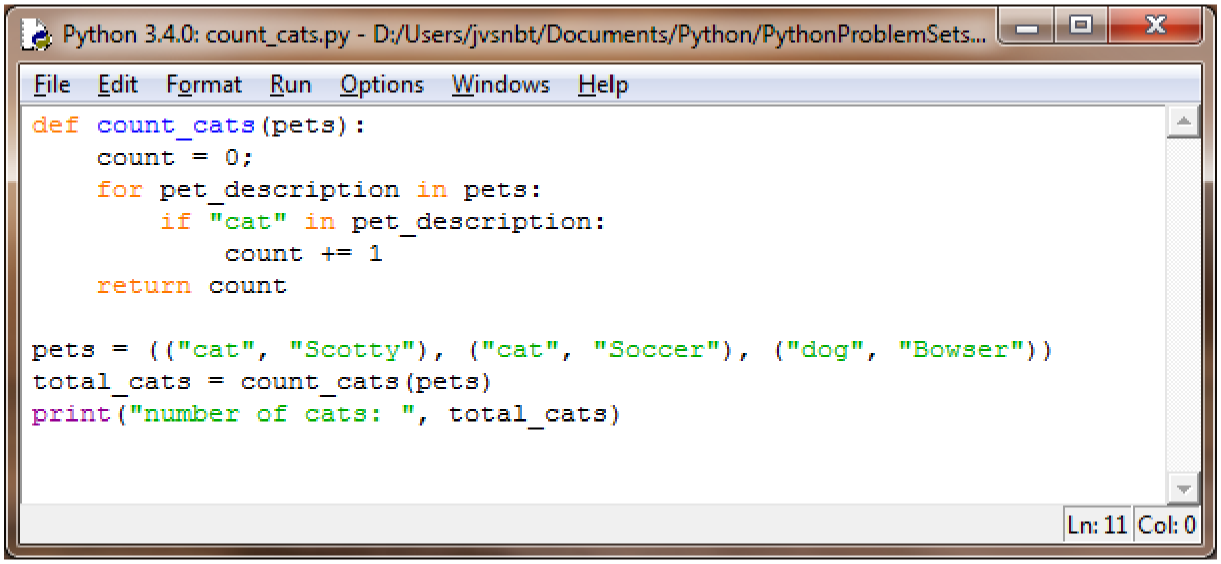
Since the variable named pets is a tuple and thus also a sequence, the command len(pets) will return the count of items in the tuple. The answer is 3 because pets contains three tuples, one for each of the three pets: Scotty, Soccer, and Bowser.



```
>>> for letter in "hello":
	print(letter)

	
h
e
l
l
o
>>> 
```

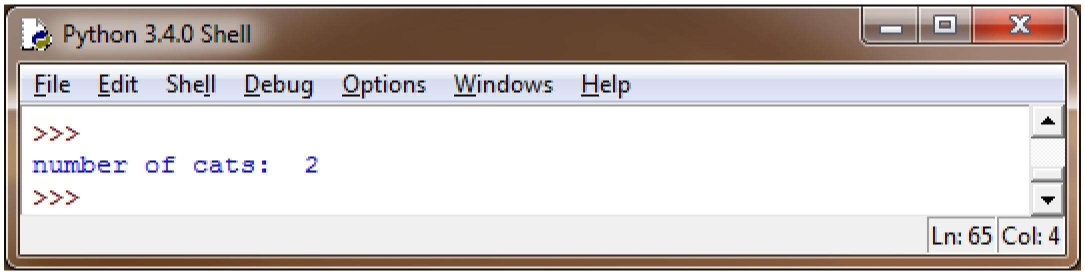
Likewise the for loop shown above prints each letter in the string “hello.” Both strings and tuples are examples of sequences, so the **for** and **in** operators work with both strings and tuples.

Our next function will count the number of cats in the collection of pets.


Python 3.4.0: count_cats.py - D:/Users/jvsnbt/Documents/Python/PythonProblemSets...
File Edit Format Run Options Windows Help
Ln: 11 Col: 0

```
def count_cats(pets):
    count = 0;
    for pet_description in pets:
        if "cat" in pet_description:
            count += 1
    return count

pets = (("cat", "Scotty"), ("cat", "Soccer"), ("dog", "Bowser"))
total_cats = count_cats(pets)
print("number of cats: ", total_cats)
```

When run count_cats displayed a total of 2.



```
>>>
number of cats:  2
>>>
```

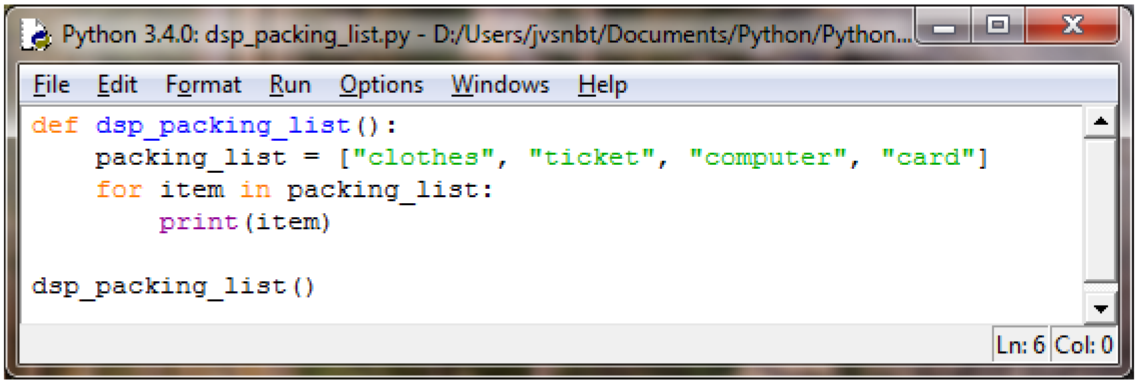
When processing all the items in a sequence our python functions used the pattern of “**for** item **in** sequence.” Because not all the items were cats our current python function included an **if** operator within the loop.

Exercises:

Topic: Lists

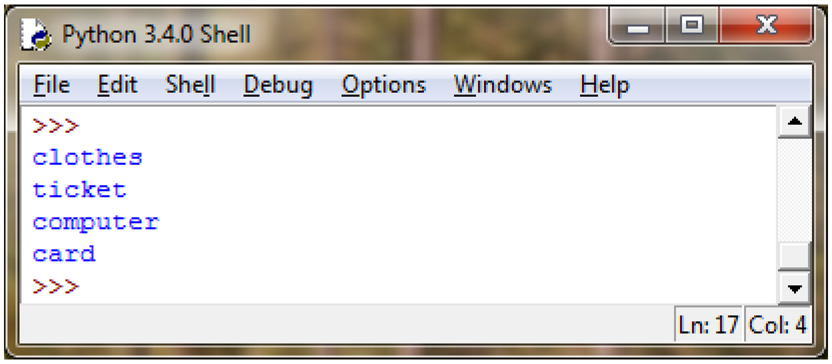
A list is a mutable sequence of items. Mutable means the list can change. Lists begin and end with square brackets [and]. An example is a packing list for a trip. The items to pack could be summarized as pack = [“clothes”, “cash”, “computer”, “credit card”]

Aside: Wikipedia describes the terminology for programming symbols at http://en.wikipedia.org/wiki/Bracket#Braces.


Python 3.4.0: dsp_packing_list.py - D:/Users/jvsnbt/Documents/Python/Python...
File Edit Format Run Options Windows Help
Ln: 6 Col: 0

```
def dsp_packing_list():
    packing_list = ["clothes", "ticket", "computer", "card"]
    for item in packing_list:
        print(item)

dsp_packing_list()
```

This python function displays the packing list.



```
>>>
clothes
ticket
computer
card
>>>
```

A python computer application to maintain that list would allow the user to display, update, and sort the list. Notice the comment starting with a hash character, which is the # symbol described by Wikipedia at http://en.wikipedia.org/wiki/Number_sign. Next the **global** variable named **needs** is defined outside of any function. Any function using that variable must use a **global statement** before its first usage of that variable. For example the dsp_needs function will display a list of every item in the list named needs, so that function started with the global statement: **global needs**. Then that function can use the list named **needs**.

The next function, named **update_needs** will maintain the list of needs. The user is presented with three choices. Choice **a** will add a new item to the list of needs. Choice **s** will show all the items in the list of needs. Choice **d** will delete an item. The input command displays a prompt and accepts the user's choice. That choice is stored in the variable named **command**.

Aside: Of course **choice** would have been a better name than **command**. Using well chosen names for variables helps programmers comprehend your program.

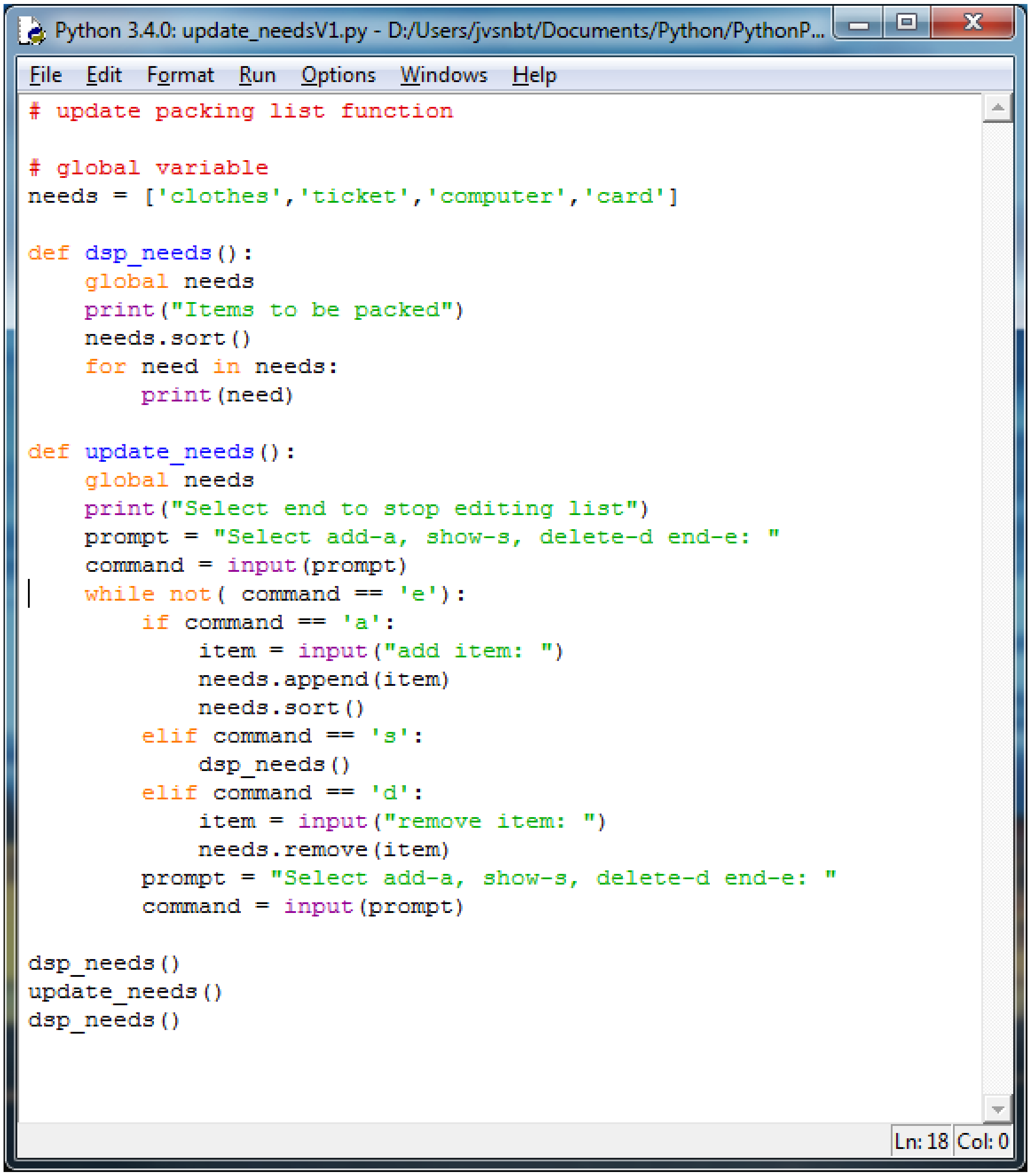
The function then needs to set up a loop, so the function will perform each of the user's commands until a command of **e** is input. The **while** command repeats a series of commands while the loop criteria is satisfied. Those commands are indented below the while command. When the user enters an e command to end execution then the statement **command == 'e'** is **True**. Our program should continue looping while that statement is **False**, so the loop criteria for our example is **not (command == 'e')**. Thus our while statement is written as **while not (command == 'e')**.

An equivalent command uses the **!=** operator, which means **not equal**. Then the **while** command is written as **while command != 'e'**.

Of course the last statement in the while loop asks for the user's next command.

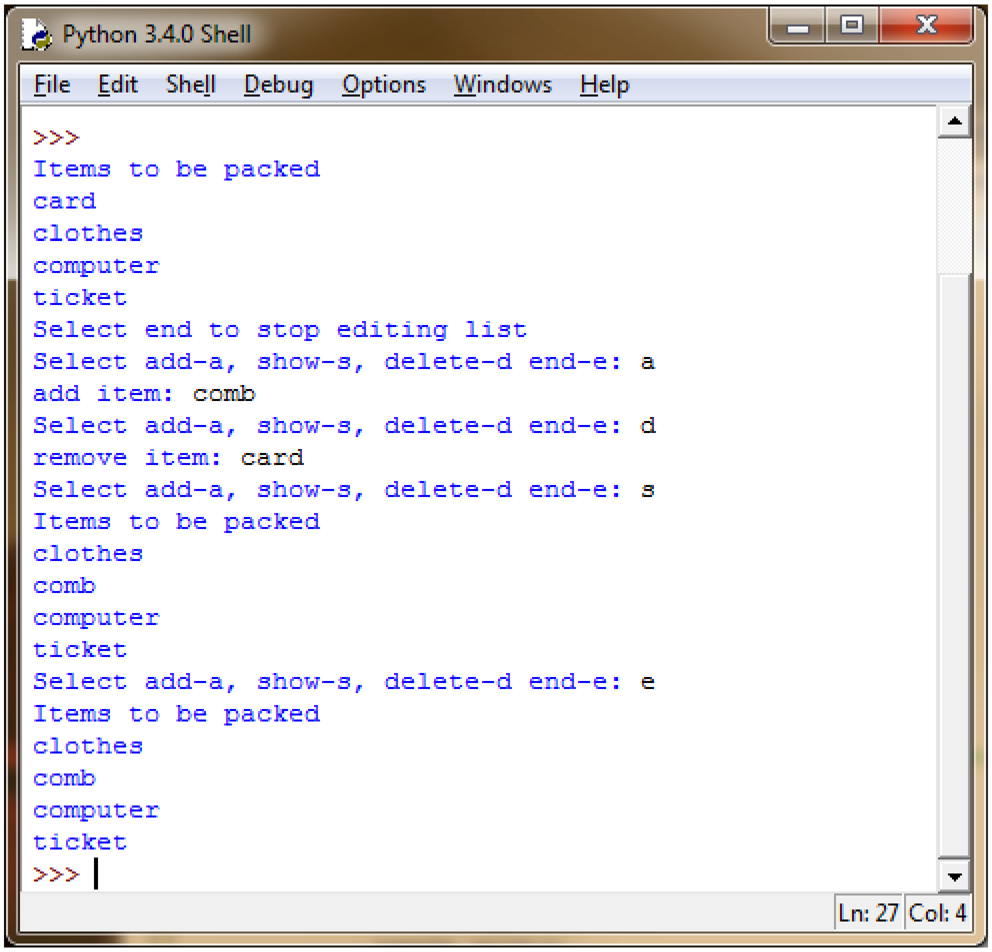
Basic structure of a while loop
1. command = input(prompt)
2. while command != end
3. execute series of commands
4. command = input(prompt)

Inside the **while** statement the function uses a series of **if** statements and **elif** statements to process the user's command. Including an **else** statement would allow for a response to any incorrect command such as Z.


Python 3.4.0: update_needsV1.py - D:/Users/jvsnbt/Documents/Python/PythonP...
File Edit Format Run Options Windows Help
Ln: 18 Col: 0

```
# update packing list function

# global variable
needs = ['clothes','ticket','computer','card']

def dsp_needs():
    global needs
    print("Items to be packed")
    needs.sort()
    for need in needs:
        print(need)

def update_needs():
    global needs
    print("Select end to stop editing list")
    prompt = "Select add-a, show-s, delete-d end-e: "
    command = input(prompt)
    while not( command == 'e'):
        if command == 'a':
            item = input("add item: ")
            needs.append(item)
            needs.sort()
        elif command == 's':
            dsp_needs()
        elif command == 'd':
            item = input("remove item: ")
            needs.remove(item)
        prompt = "Select add-a, show-s, delete-d end-e: "
        command = input(prompt)

dsp_needs()
update_needs()
dsp_needs()
```

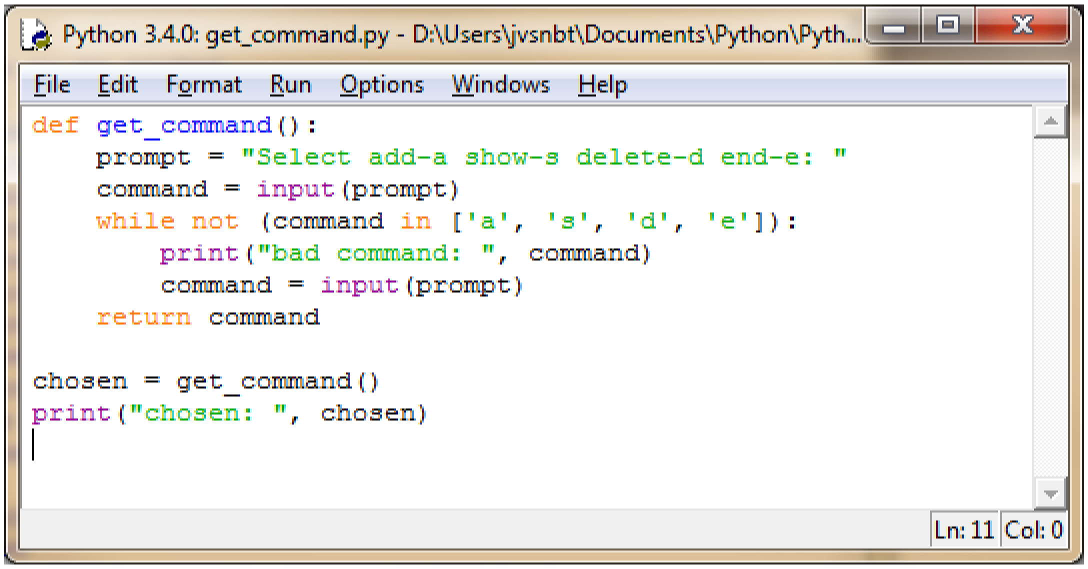
A practice execution of this program to add a comb and remove the credit card resulted in this display.



```
>>> 
Items to be packed
card
clothes
computer
ticket
Select end to stop editing list
Select add-a, show-s, delete-d end-e: a
add item: comb
Select add-a, show-s, delete-d end-e: d
remove item: card
Select add-a, show-s, delete-d end-e: s
Items to be packed
clothes
comb
computer
ticket
Select add-a, show-s, delete-d end-e: e
Items to be packed
clothes
comb
computer
ticket
>>> 
```

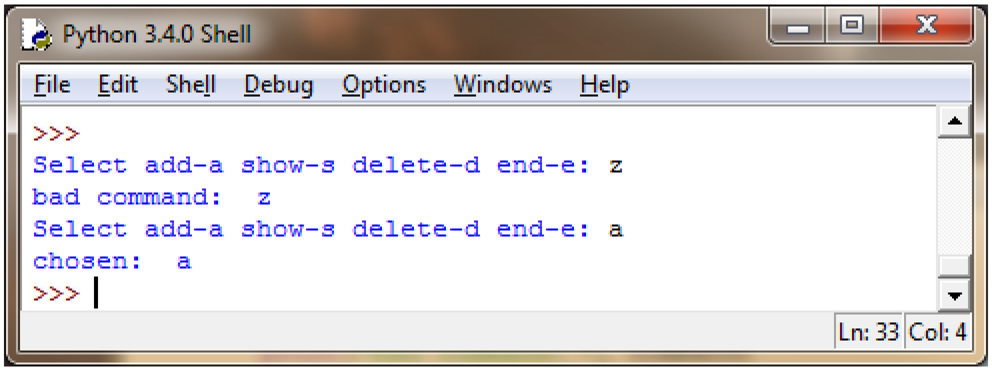
Topic: An improvement to the program, verify the input.

An improvement to the user interface would check that the user's input is a correct code. The codes of **a, s, d,** and **e** are acceptable, but **z** is not. Immediately following the input statement, the program should check that the command is in the list of acceptable commands, **['a', 's', 'd', 'e']**. Because the user could repeatedly input bad codes a loop is needed. Again a **while** loop is used.


Python 3.4.0: get_command.py - D:\Users\jvsnbt\Documents\Python\Pyth...
File Edit Format Run Options Windows Help
Ln: 11 Col: 0

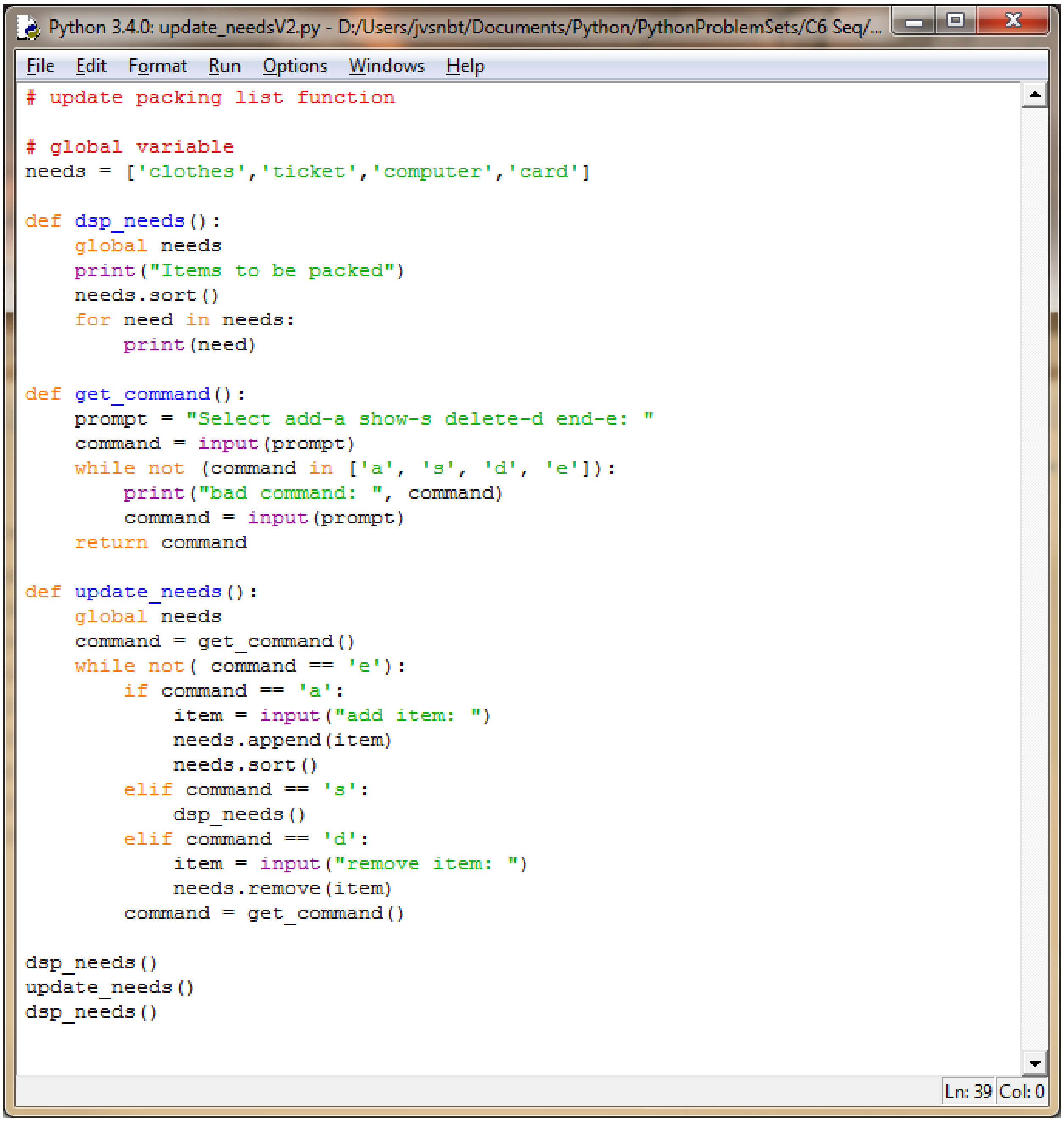
```
def get_command():
    prompt = "Select add-a show-s delete-d end-e: "
    command = input(prompt)
    while not (command in ['a', 's', 'd', 'e']):
        print("bad command: ", command)
        command = input(prompt)
    return command

chosen = get_command()
print("chosen: ", chosen)
```

This test execution tried an unacceptable code of **z** followed by an input command of **a**. The function displayed the expected results.



```
>>>
Select add-a show-s delete-d end-e: z
bad command:  z
Select add-a show-s delete-d end-e: a
chosen:  a
>>>
```

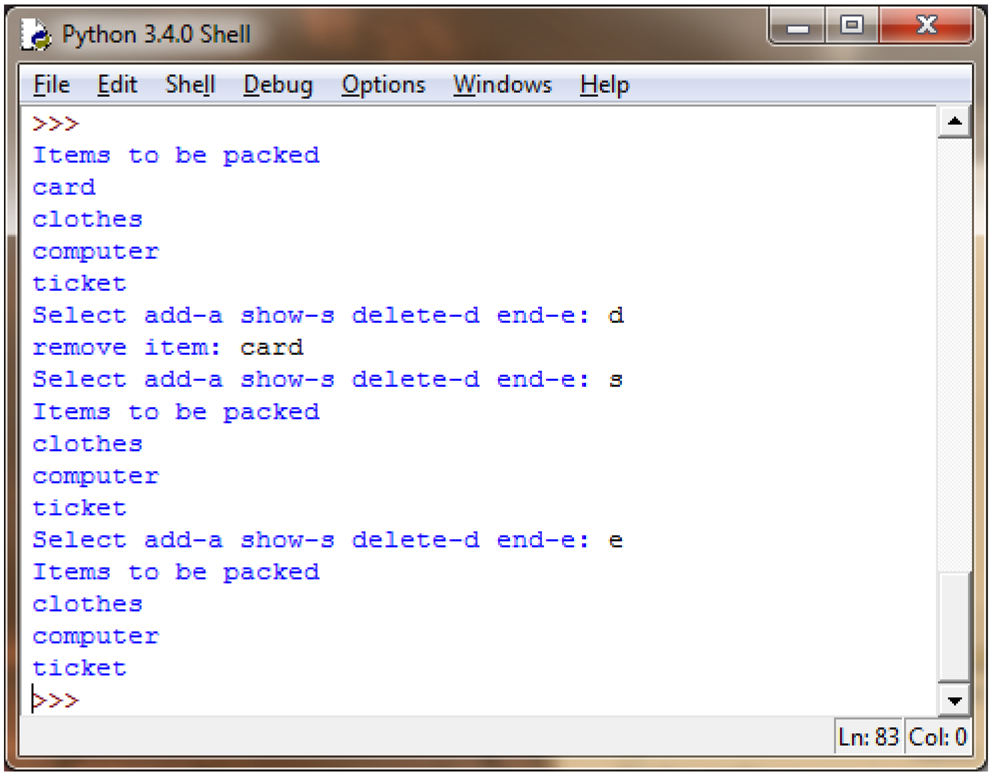
Next the new function will be integrated into the main program. Because the input command is used in two places the input commands should be replaced with a call to the new functions. Also the new function contains a **while** loop, so keeping the new function separate will avoid having a **while** loop contained within the **update_needs while** loop. Thus the **update_needs** function remains simple and easy to comprehend.


Python 3.4.0: update_needsV2.py - D:/Users/jvsnbt/Documents/Python/PythonProblemSets/C6 Seq/...

```
# update packing list function

# global variable
needs = ['clothes','ticket','computer','card']

def dsp_needs():
    global needs
    print("Items to be packed")
    needs.sort()
    for need in needs:
        print(need)

def get_command():
    prompt = "Select add-a show-s delete-d end-e: "
    command = input(prompt)
    while not (command in ['a', 's', 'd', 'e']):
        print("bad command: ", command)
        command = input(prompt)
    return command

def update_needs():
    global needs
    command = get_command()
    while not( command == 'e'):
        if command == 'a':
            item = input("add item: ")
            needs.append(item)
            needs.sort()
        elif command == 's':
            dsp_needs()
        elif command == 'd':
            item = input("remove item: ")
            needs.remove(item)
        command = get_command()

dsp_needs()
update_needs()
dsp_needs()
```

Ln: 39 Col: 0

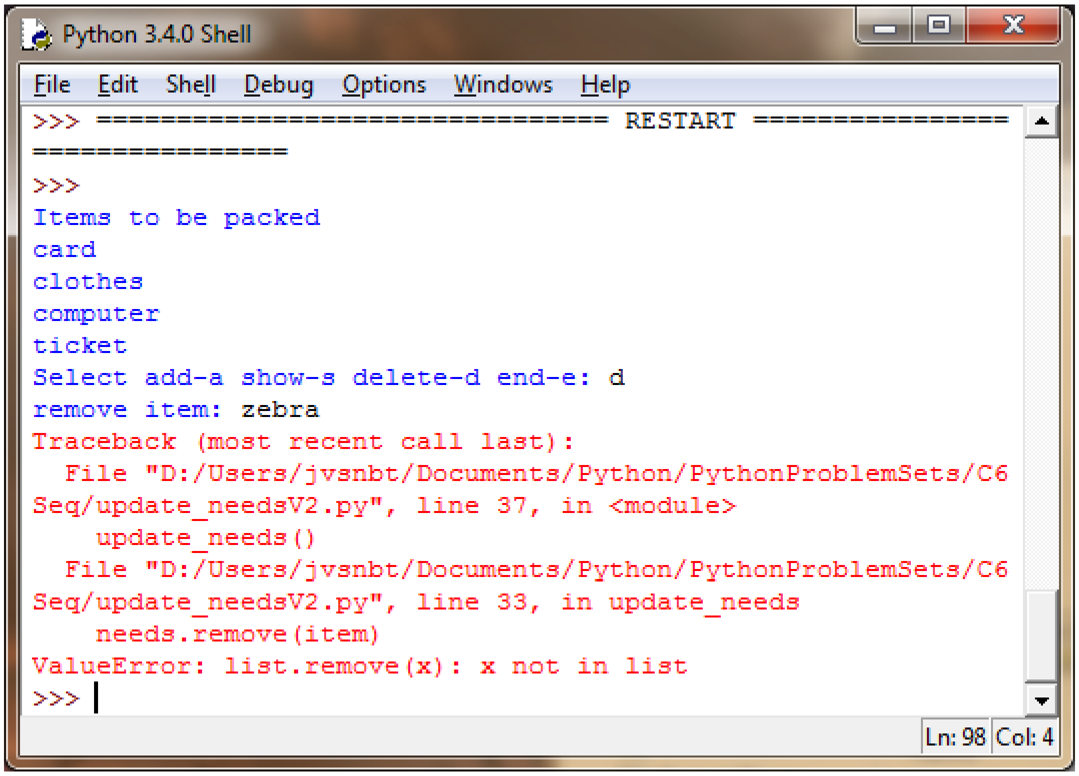
A simple test run of this program will remove the card and end the run.



```
>>>
Items to be packed
card
clothes
computer
ticket
Select add-a show-s delete-d end-e: d
remove item: card
Select add-a show-s delete-d end-e: s
Items to be packed
clothes
computer
ticket
Select add-a show-s delete-d end-e: e
Items to be packed
clothes
computer
ticket
>>>
```

Future project: Handling execution errors

Of course the user should not attempt to delete an item not in the list. For example attempting to remove a zebra caused an error, so IDLE stopped the program's execution an displayed an error message.

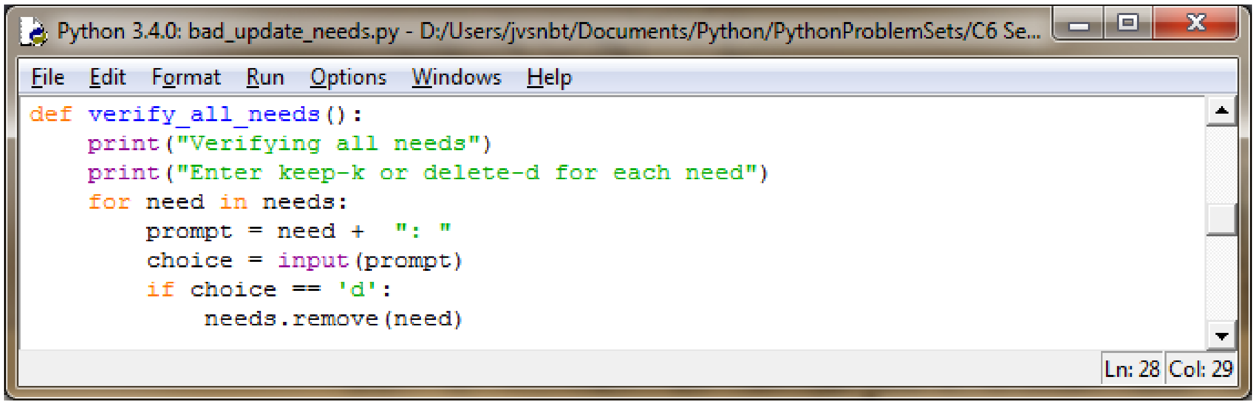


```
>>> ================================= RESTART ================================
>>>
Items to be packed
card
clothes
computer
ticket
Select add-a show-s delete-d end-e: d
remove item: zebra
Traceback (most recent call last):
  File "D:/Users/jvsnbt/Documents/Python/PythonProblemSets/C6Seq/update_needsV2.py", line 37, in <module>
    update_needs()
  File "D:/Users/jvsnbt/Documents/Python/PythonProblemSets/C6Seq/update_needsV2.py", line 33, in update_needs
    needs.remove(item)
ValueError: list.remove(x): x not in list
>>>
```

Later chapters will explain methods for handling such execution errors.

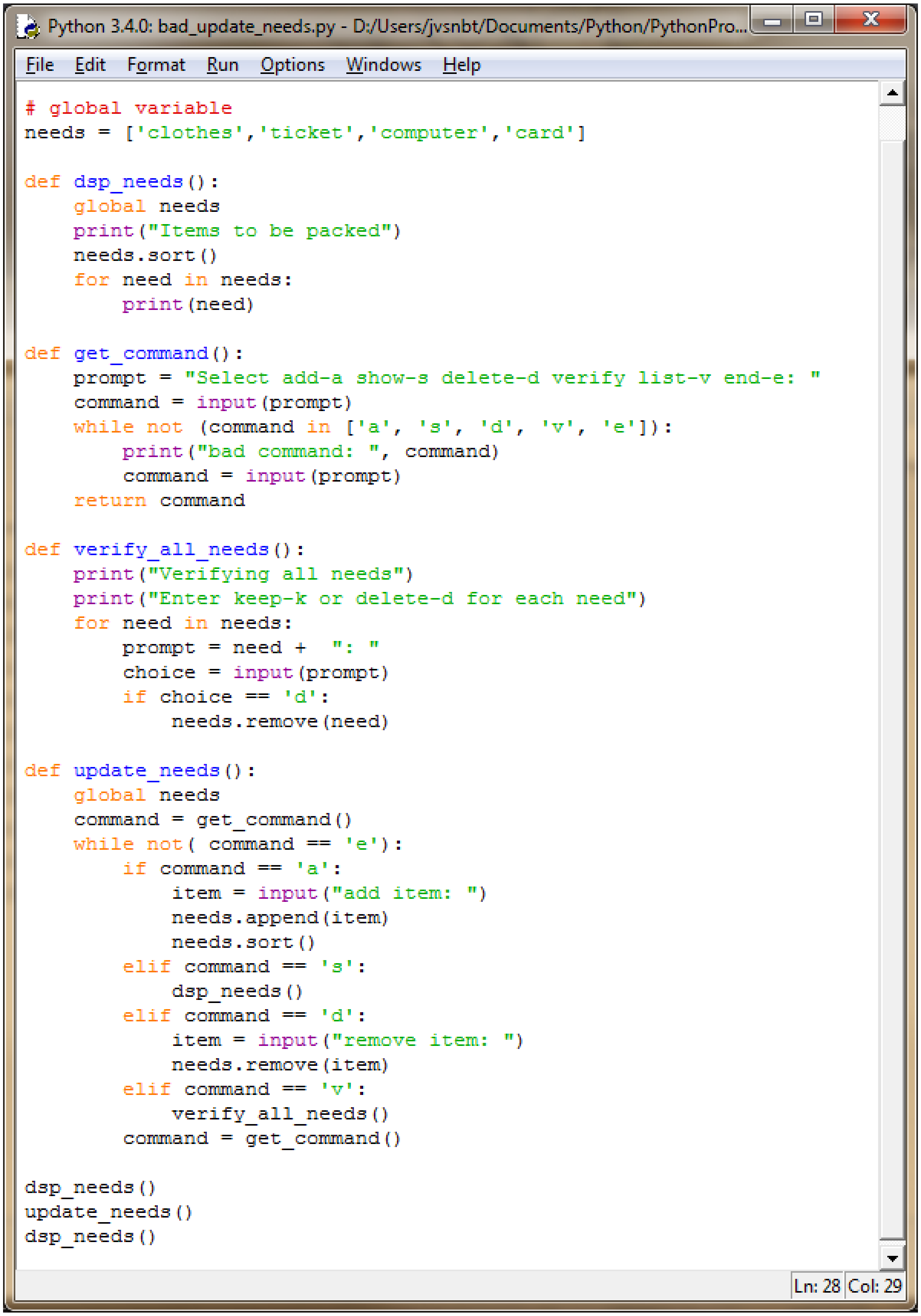
Warning: Do not alter a list while looping through each item.

The program encourages users to show the entire list after each update command. No one wants to wait for large lists to be repeatedly displayed. Thus a programmer writing a function to verify each item would be tempted to include the **if** and **elif** statements within a **for-each** loop. The result is disastrous.


Python 3.4.0: bad_update_needs.py - D:/Users/jvsnbt/Documents/Python/PythonProblemSets/C6 Se...
File Edit Format Run Options Windows Help
Ln: 28 Col: 29

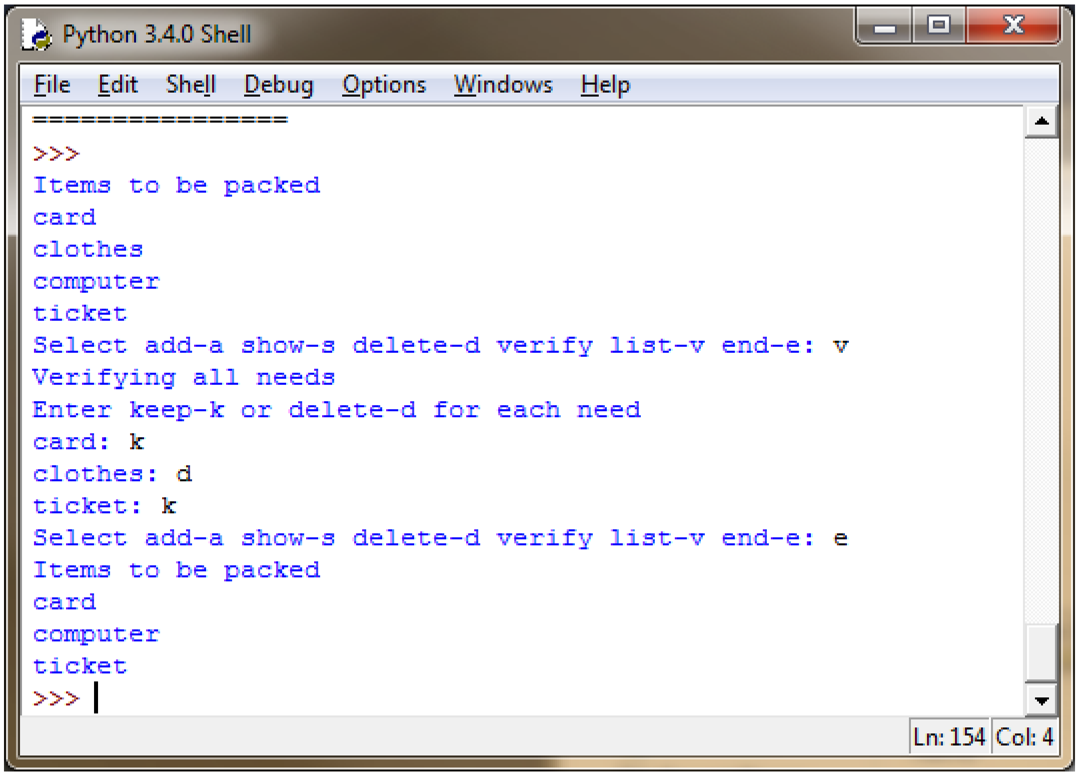
```python
def verify_all_needs():
    print("Verifying all needs")
    print("Enter keep-k or delete-d for each need")
    for need in needs:
        prompt = need +  ": "
        choice = input(prompt)
        if choice == 'd':
            needs.remove(need)
```

Also the main program should be modified to include this new **verify_all_n eeds** function.


Python 3.4.0: bad_update_needs.py - D:/Users/jvsnbt/Documents/Python/PythonPro...
File Edit Format Run Options Windows Help
Ln: 28 Col: 29

```python
# global variable
needs = ['clothes','ticket','computer','card']

def dsp_needs():
    global needs
    print("Items to be packed")
    needs.sort()
    for need in needs:
        print(need)

def get_command():
    prompt = "Select add-a show-s delete-d verify list-v end-e: "
    command = input(prompt)
    while not (command in ['a', 's', 'd', 'v', 'e']):
        print("bad command: ", command)
        command = input(prompt)
    return command

def verify_all_needs():
    print("Verifying all needs")
    print("Enter keep-k or delete-d for each need")
    for need in needs:
        prompt = need +  ": "
        choice = input(prompt)
        if choice == 'd':
            needs.remove(need)

def update_needs():
    global needs
    command = get_command()
    while not( command == 'e'):
        if command == 'a':
            item = input("add item: ")
            needs.append(item)
            needs.sort()
        elif command == 's':
            dsp_needs()
        elif command == 'd':
            item = input("remove item: ")
            needs.remove(item)
        elif command == 'v':
            verify_all_needs()
        command = get_command()

dsp_needs()
update_needs()
dsp_needs()
```

A test execution initially appears to be good until you notice that the computer need was never presented to the user for verification.



```
================
>>>
Items to be packed
card
clothes
computer
ticket
Select add-a show-s delete-d verify list-v end-e: v
Verifying all needs
Enter keep-k or delete-d for each need
card: k
clothes: d
ticket: k
Select add-a show-s delete-d verify list-v end-e: e
Items to be packed
card
computer
ticket
>>> 
```

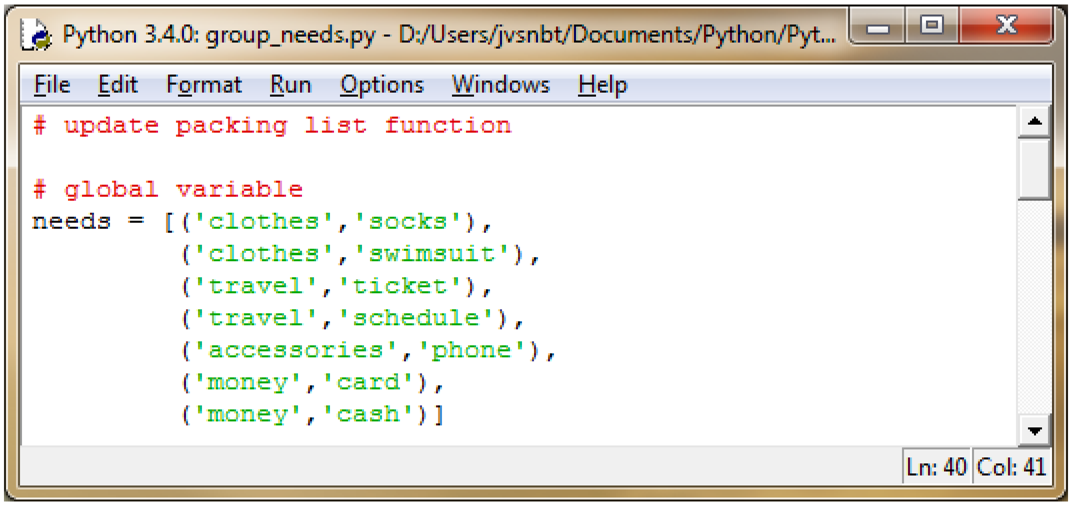
The loop was initially set up to progress through all items in the list from **needs[0]** through to **needs[3]**. **Needs[0]** elicited a response of keep, so the loop progressed forward to **needs[1]** which is clothes. The user deleted the clothes item from the list. The **remove** method used to remove clothes from the list changes the original list. The loop command is not aware of any change. The loop command has completed **needs[0]** and **needs[1]**, so the next item is **needs[2]** which is now ticket, not computer as it had been. Thus computer which moved to **needs[1]** was never presented to the user for verification.

Index	Original list	Altered list

0	Card	Card
1	Clothes	Computer
2	Computer	Ticket
3	Ticket	

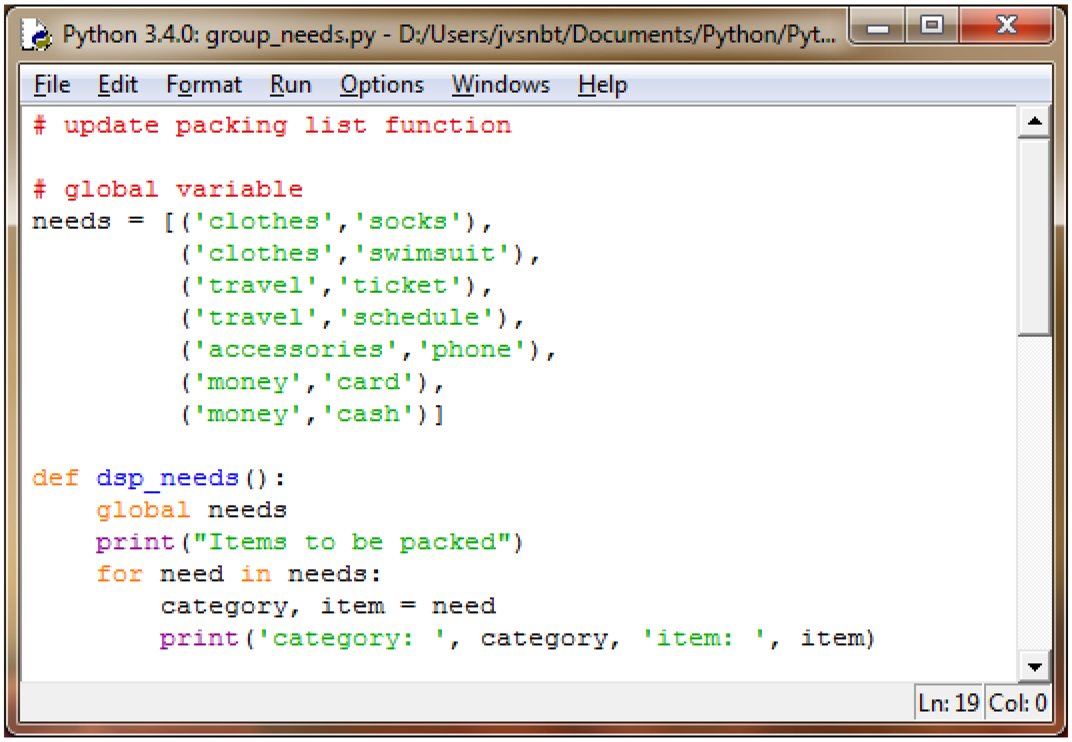
Topic: Categorize each need – Use a list of tuples

Each item can be categorized. For example the ticket item could be in the category of transportation. Tuples allow multiple attributes to be together as an immutable object. For each the ticket item could be in a tuple as **('transportation', 'ticket')**. The list of travel needs could then be represented as a list of tuples.


Python 3.4.0: group_needs.py - D:/Users/jvsnbt/Documents/Python/Pyt...
File Edit Format Run Options Windows Help
Ln: 40 Col: 41

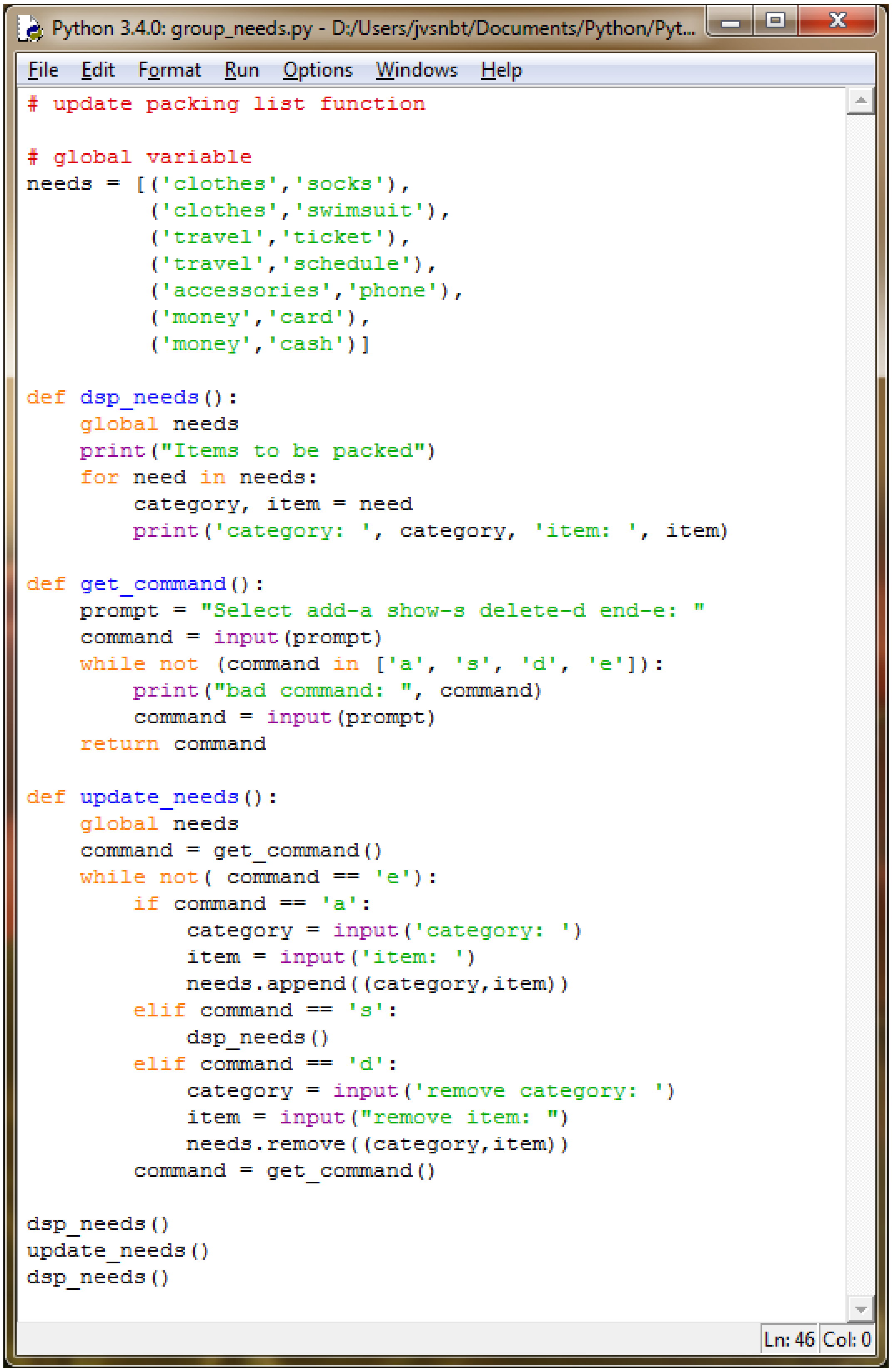
```
# update packing list function

# global variable
needs = [('clothes','socks'),
         ('clothes','swimsuit'),
         ('travel','ticket'),
         ('travel','schedule'),
         ('accessories','phone'),
         ('money','card'),
         ('money','cash')]
```

The print statement within the dsp_needs function needs to be altered for displaying a tuple. An example of sequence packing is **t = 'clothes', 'socks'** which is equivalent to the command to create a tuple such as **('clothes', 'socks')** and assign that tuple to variable **t**. That statement is written as **t =** (**'clothes', 'socks')**. Sequence unpacking works reverses that packing statement. To unpack the category and item from a need the statement is **attribute1, attribute2, attribute3 = tuple** where exactly the same number of variables are needs as the tuple contains attributes. For our program the sequence unpacking statement is **category, item = need**. Then the print statement can print each component of the tuple named **need**.


Python 3.4.0: group_needs.py - D:/Users/jvsnbt/Documents/Python/Pyt...
File Edit Format Run Options Windows Help
Ln: 19 Col: 0

```
# update packing list function

# global variable
needs = [('clothes','socks'),
         ('clothes','swimsuit'),
         ('travel','ticket'),
         ('travel','schedule'),
         ('accessories','phone'),
         ('money','card'),
         ('money','cash')]

def dsp_needs():
    global needs
    print("Items to be packed")
    for need in needs:
        category, item = need
        print('category: ', category, 'item: ', item)
```

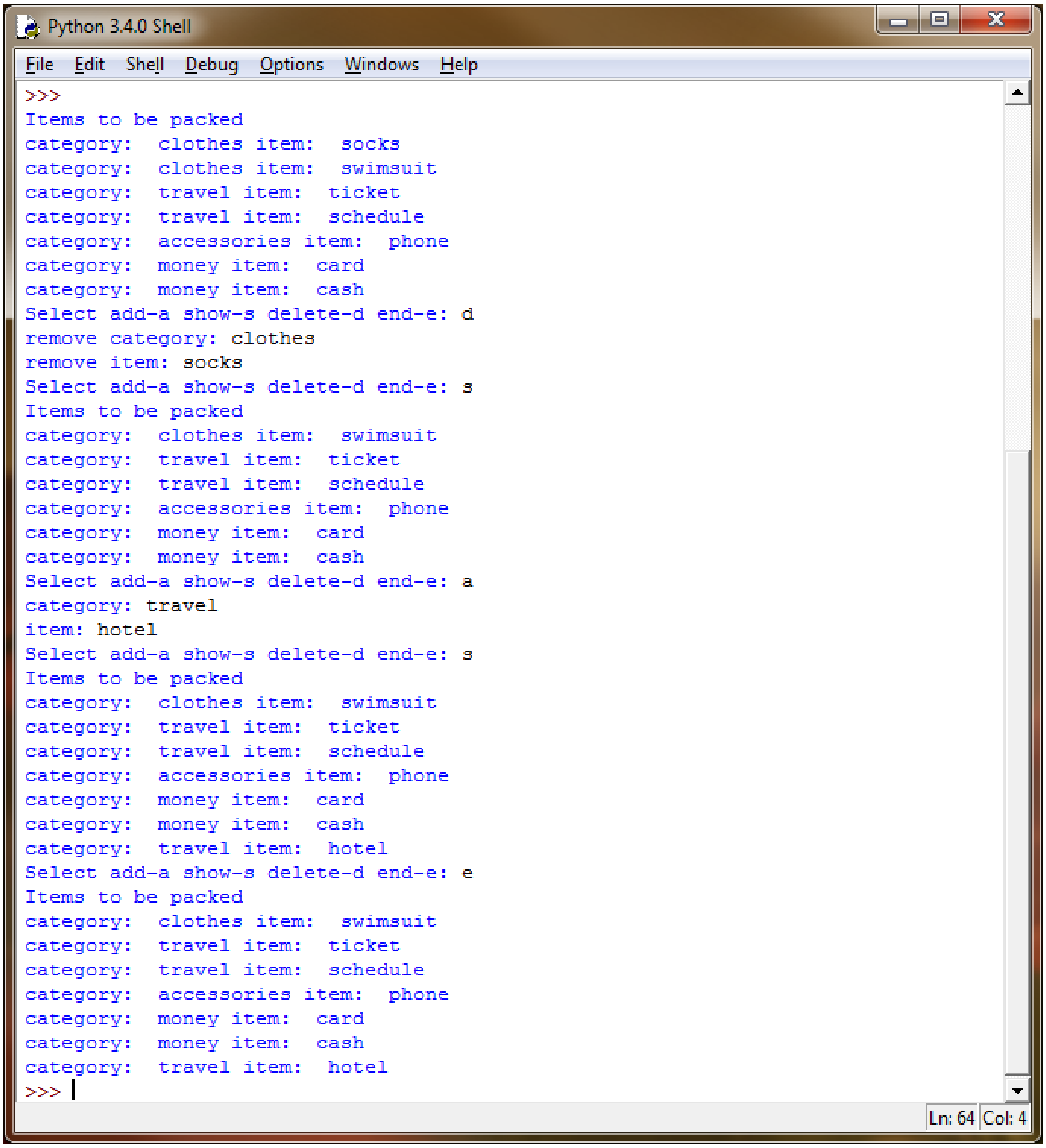
Finally the update_needs function must request both the category and need for any need to be added or removed from the list.



Python 3.4.0: group_needs.py - D:/Users/jvsnbt/Documents/Python/Pyt...
File Edit Format Run Options Windows Help
Ln: 46 Col: 0

```
# update packing list function

# global variable
needs = [('clothes','socks'),
         ('clothes','swimsuit'),
         ('travel','ticket'),
         ('travel','schedule'),
         ('accessories','phone'),
         ('money','card'),
         ('money','cash')]

def dsp_needs():
    global needs
    print("Items to be packed")
    for need in needs:
        category, item = need
        print('category: ', category, 'item: ', item)

def get_command():
    prompt = "Select add-a show-s delete-d end-e: "
    command = input(prompt)
    while not (command in ['a', 's', 'd', 'e']):
        print("bad command: ", command)
        command = input(prompt)
    return command

def update_needs():
    global needs
    command = get_command()
    while not( command == 'e'):
        if command == 'a':
            category = input('category: ')
            item = input('item: ')
            needs.append((category,item))
        elif command == 's':
            dsp_needs()
        elif command == 'd':
            category = input('remove category: ')
            item = input("remove item: ")
            needs.remove((category,item))
        command = get_command()

dsp_needs()
update_needs()
dsp_needs()
```

A test execution of this program deleted socks and added hotel reservations.



```
>>>
Items to be packed
category:  clothes item:  socks
category:  clothes item:  swimsuit
category:  travel item:  ticket
category:  travel item:  schedule
category:  accessories item:  phone
category:  money item:  card
category:  money item:  cash
Select add-a show-s delete-d end-e: d
remove category: clothes
remove item: socks
Select add-a show-s delete-d end-e: s
Items to be packed
category:  clothes item:  swimsuit
category:  travel item:  ticket
category:  travel item:  schedule
category:  accessories item:  phone
category:  money item:  card
category:  money item:  cash
Select add-a show-s delete-d end-e: a
category: travel
item: hotel
Select add-a show-s delete-d end-e: s
Items to be packed
category:  clothes item:  swimsuit
category:  travel item:  ticket
category:  travel item:  schedule
category:  accessories item:  phone
category:  money item:  card
category:  money item:  cash
category:  travel item:  hotel
Select add-a show-s delete-d end-e: e
Items to be packed
category:  clothes item:  swimsuit
category:  travel item:  ticket
category:  travel item:  schedule
category:  accessories item:  phone
category:  money item:  card
category:  money item:  cash
category:  travel item:  hotel
>>> 
```

Fun for all majors:

Step 1: Maintain a list of upcoming movie titles that you want to see.

Step 2: Instead of simply the title, use a tuple to store the movie's category, title, and release date)

Computer Science:

Count the occurrences of a letter within a string. This is a basic starting strategy for code cracking. The frequency of the letters, such as “e,” depends on the language.

Business:

Process a list of stock sales and purchases. Each sale is stored as a tuple containing the sale date stored as YYMMDD, the company's NYSE abbreviation, number of shares, and total price of the transaction. Ask the user which day and company to process. Display the total shares sold or purchased.

Software Engineering:

Maintain a list of the data sources for a project. The list hold tuples. Each tuple holds the abbreviated name of the data source, field names within each data source, data type, and description of each field.

Biotechnology:

DNA, described by Wikipedia at http://en.wikipedia.org/wiki/DNA, consists of a series of letters. Write a python function called chk_DNA_letters to check that a list of such DNA letters stored as strings contains only those appropriate DNA letters. Display only the good strings. Display a count of the number of good strings.

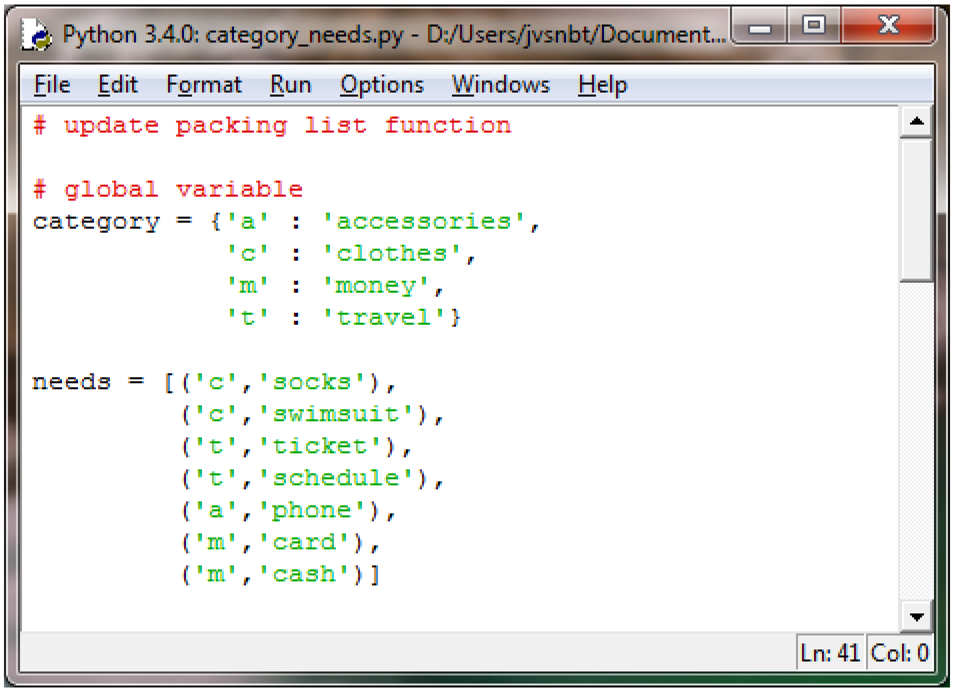
Mathematics:

Write an program to hold a finite series containing real numbers. Display a running total of the series as you proceed through the series.

Chapter 7: Dictionaries

Topic: Dictionaries

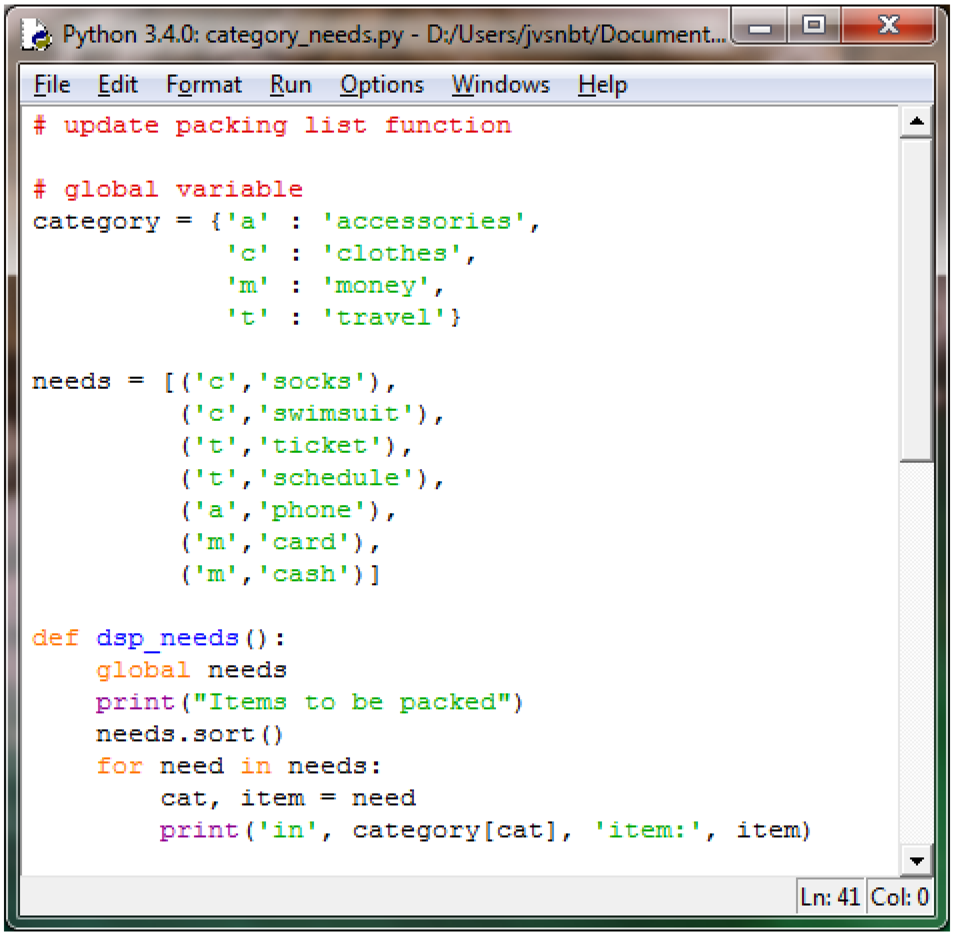
A dictionary stores pairs of values. For example our program for tracking travel needs would benefit from a dictionary of the item categories and their single letter abbreviations.


Python 3.4.0: category_needs.py - D:/Users/jvsnbt/Document...
File Edit Format Run Options Windows Help
Ln: 41 Col: 0

```
# update packing list function

# global variable
category = {'a' : 'accessories',
            'c' : 'clothes',
            'm' : 'money',
            't' : 'travel'}

needs = [('c','socks'),
         ('c','swimsuit'),
         ('t','ticket'),
         ('t','schedule'),
         ('a','phone'),
         ('m','card'),
         ('m','cash')]
```

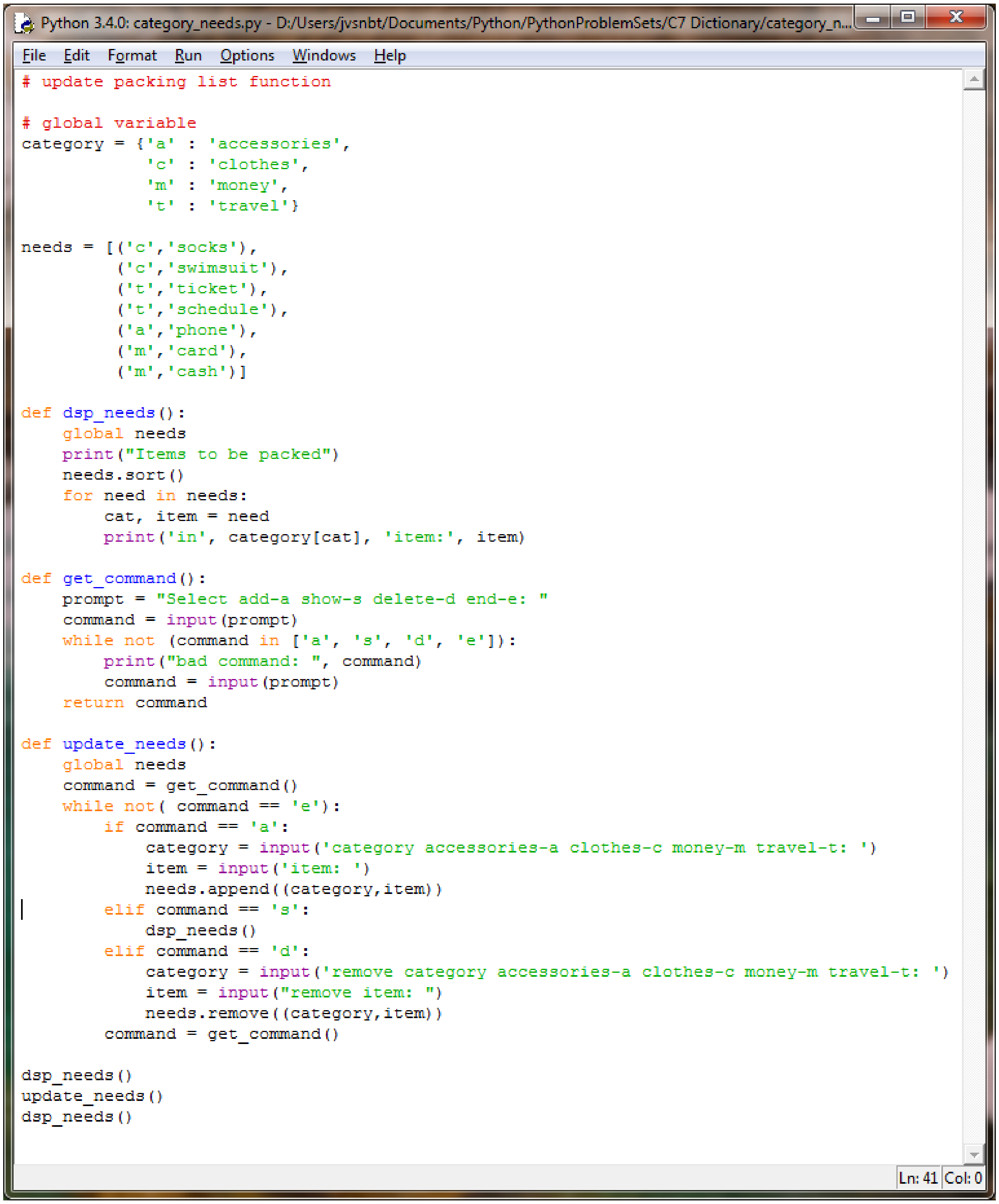
Now the list of needs is shorter since only a single letter is stored as the category, not the entire word. When printing the list of needs the program prints **category[cat]** where cat is the key used in the dictionary of categories.

The dictionary named category has its key values in ascending order, but such order is not required. The key and value pairs can be in any order within a dictionary. No order is assumed, so an index value of 0 would produce an error. For example **category[0]** would return an error because the key value of 0 is not in the dictionary named category. Dictionaries do not behave like lists.


Python 3.4.0: category_needs.py - D:/Users/jvsnbt/Document...
File Edit Format Run Options Windows Help
Ln: 41 Col: 0

```
# update packing list function

# global variable
category = {'a' : 'accessories',
            'c' : 'clothes',
            'm' : 'money',
            't' : 'travel'}

needs = [('c','socks'),
         ('c','swimsuit'),
         ('t','ticket'),
         ('t','schedule'),
         ('a','phone'),
         ('m','card'),
         ('m','cash')]

def dsp_needs():
    global needs
    print("Items to be packed")
    needs.sort()
    for need in needs:
        cat, item = need
        print('in', category[cat], 'item:', item)
```

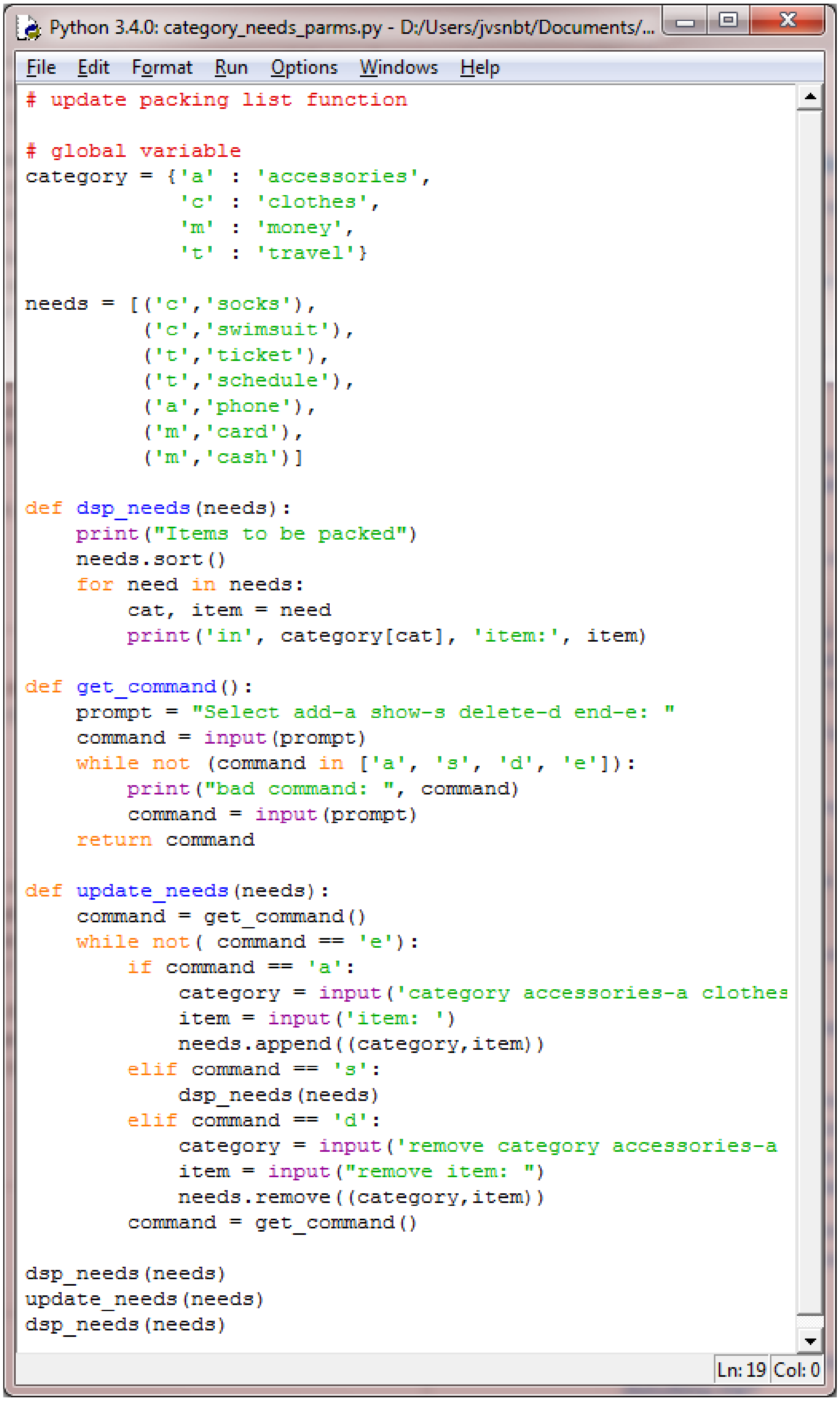
When entering or deleting an item the user now can type the abbreviation instead of the full name of the category. The update function remains unchanged. The variable named **category** now accepts the single letter code of the category. That new abbreviated category code and the item's description now can be used to update the list of items in the list named **needs**.


Python 3.4.0: category_needs.py - D:/Users/jvsnbt/Documents/Python/PythonProblemSets/C7 Dictionary/category_n...
File Edit Format Run Options Windows Help
Ln: 41 Col: 0

```python
# update packing list function

# global variable
category = {'a' : 'accessories',
            'c' : 'clothes',
            'm' : 'money',
            't' : 'travel'}

needs = [('c','socks'),
         ('c','swimsuit'),
         ('t','ticket'),
         ('t','schedule'),
         ('a','phone'),
         ('m','card'),
         ('m','cash')]

def dsp_needs():
    global needs
    print("Items to be packed")
    needs.sort()
    for need in needs:
        cat, item = need
        print('in', category[cat], 'item:', item)

def get_command():
    prompt = "Select add-a show-s delete-d end-e: "
    command = input(prompt)
    while not (command in ['a', 's', 'd', 'e']):
        print("bad command: ", command)
        command = input(prompt)
    return command

def update_needs():
    global needs
    command = get_command()
    while not( command == 'e'):
        if command == 'a':
            category = input('category accessories-a clothes-c money-m travel-t: ')
            item = input('item: ')
            needs.append((category,item))
        elif command == 's':
            dsp_needs()
        elif command == 'd':
            category = input('remove category accessories-a clothes-c money-m travel-t: ')
            item = input("remove item: ")
            needs.remove((category,item))
        command = get_command()

dsp_needs()
update_needs()
dsp_needs()
```

Topic: Avoiding global variables

Some python programmers prefer to avoid global variables. They pass the variable to each function.


Python 3.4.0: category_needs_parms.py - D:/Users/jvsnbt/Documents/...
File Edit Format Run Options Windows Help
Ln: 19 Col: 0

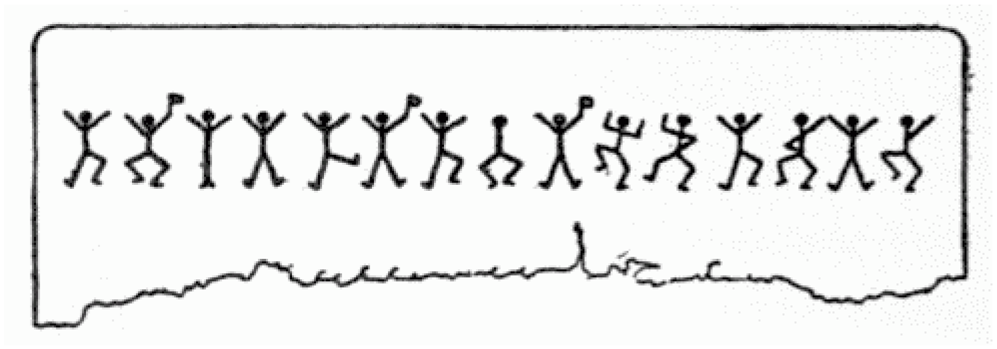
```
# update packing list function

# global variable
category = {'a' : 'accessories',
            'c' : 'clothes',
            'm' : 'money',
            't' : 'travel'}

needs = [('c','socks'),
         ('c','swimsuit'),
         ('t','ticket'),
         ('t','schedule'),
         ('a','phone'),
         ('m','card'),
         ('m','cash')]

def dsp_needs(needs):
    print("Items to be packed")
    needs.sort()
    for need in needs:
        cat, item = need
        print('in', category[cat], 'item:', item)

def get_command():
    prompt = "Select add-a show-s delete-d end-e: "
    command = input(prompt)
    while not (command in ['a', 's', 'd', 'e']):
        print("bad command: ", command)
        command = input(prompt)
    return command

def update_needs(needs):
    command = get_command()
    while not( command == 'e'):
        if command == 'a':
            category = input('category accessories-a clothes
            item = input('item: ')
            needs.append((category,item))
        elif command == 's':
            dsp_needs(needs)
        elif command == 'd':
            category = input('remove category accessories-a
            item = input("remove item: ")
            needs.remove((category,item))
        command = get_command()

dsp_needs(needs)
update_needs(needs)
dsp_needs(needs)
```

Topic: Using a dictionary to count entries

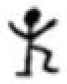
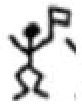
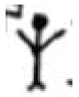
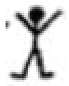
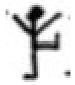
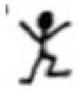
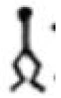
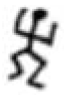
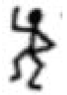
In The Adventure of the Dancing Men Sherlock Holmes is confronted my a series of messages written in symbols of dancing men. He suspects each symbol is a substitution for a single letter, so he must count how often they occur. The story is available from Gutenberg at http://www.gutenberg.org/files/108/108-h/108-h.htm#linkH2H_4_0003. The story simplifies the processing because Sherlock assumes that the flag represents the end of a word. Next Sherlock guesses that the code is a simple substitution code thus each dancing man symbol represents a letter in the English language. For further information concerning that class of code visit http://en.wikipedia.org/wiki/Substitution_cipher. How often each letter appears in English language prose has been tallied. Those totals are shown at http://en.wikipedia.org/wiki/Letter_frequency. The letter E is the most common letter in most English language correspondence. Of course the nature of the correspondence has a sever impact on the letters used in a message.

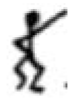
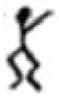
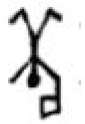
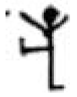
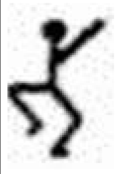
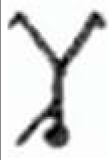
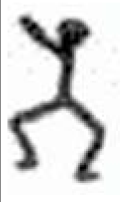
Now the reader is confronted with the boredom of tallying those symbols. Your python programming skills might help you. Convert each symbol to a number, a letter, or a name. This program chose name because people love to name the styles of dancing. Also the other options while equally valid to a computer will confuse the novice programmer. Obviously the name is not of top priority. The python programmer should simply avoid confusing names.

The first message is shown here.

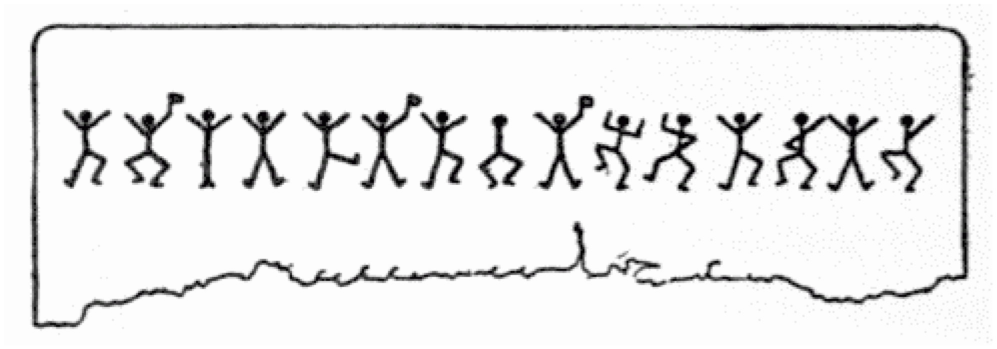


The symbols must be separated and named. One solution is shown in this table.

	rightFootUp
	clickHeels
	armsV
	spreadOut
	rightLegUp
	rightStepUp
	irish
	raveDancer
	lineDancer

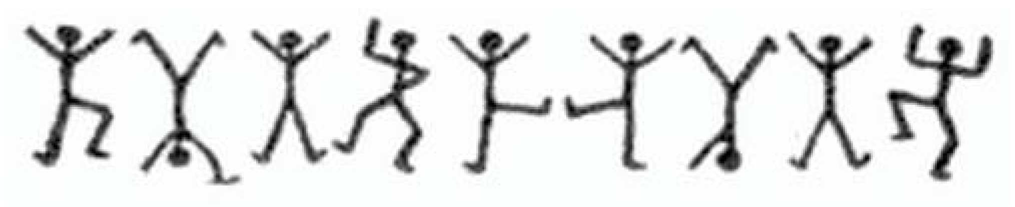
	disco
	jumpRight
	handStand
	leftLegUp
	leftLegUpRightArmOut
	headStand
	jumpLeft

How to encode message one from the criminal.

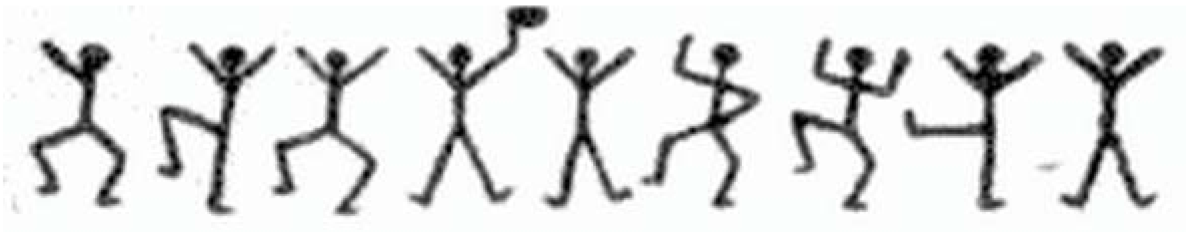


```
msg1 = [

'rightFootUp',

'clickHeels',

'armsV',

'spreadOut',

'rightLegUp',

'spreadOut',

'rightFootUp',

'irish',

'spreadOut',

'leftFootUp',

'lineDancer',

'rightFootUp',

'disco',

'spreadOut',

'leftLegUpRightArmOut']
```

Message two:

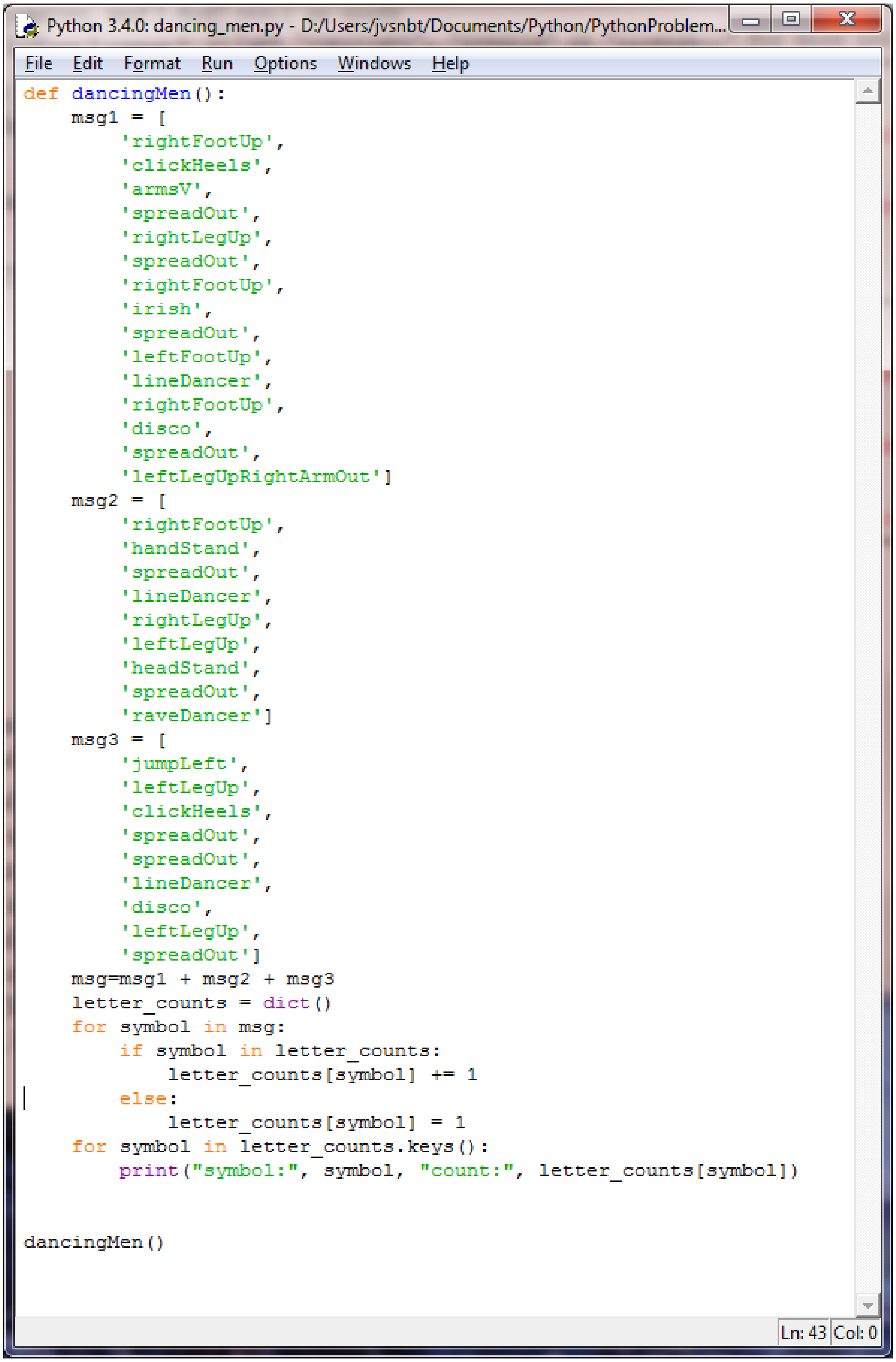


```
msg2 = [
'rightFootUp',

'handStand',

'spreadOut',

'lineDancer',

'rightLegUp',

'leftLegUp',

'headStand',

'spreadOut',

'raveDancer']
```



```
msg3 = [

'jumpLeft',

'leftLegUp',

'clickHeels',

'spreadOut',

'spreadOut',

'lineDancer',

'disco',

'leftLegUp',

'spreadOut']
```

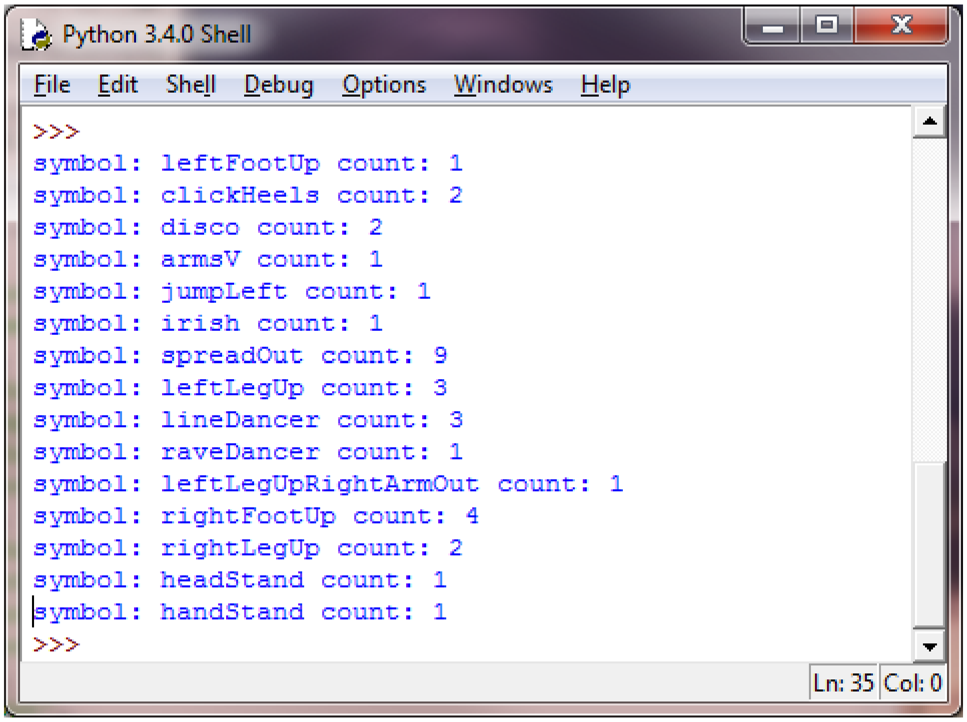
Next those messages must be counted in a dictionary in a python program.


Python 3.4.0: dancing_men.py - D:/Users/jvsnbt/Documents/Python/PythonProblem...
File Edit Format Run Options Windows Help
Ln: 43 Col: 0

```
def dancingMen():
    msg1 = [
        'rightFootUp',
        'clickHeels',
        'armsV',
        'spreadOut',
        'rightLegUp',
        'spreadOut',
        'rightFootUp',
        'irish',
        'spreadOut',
        'leftFootUp',
        'lineDancer',
        'rightFootUp',
        'disco',
        'spreadOut',
        'leftLegUpRightArmOut']
    msg2 = [
        'rightFootUp',
        'handStand',
        'spreadOut',
        'lineDancer',
        'rightLegUp',
        'leftLegUp',
        'headStand',
        'spreadOut',
        'raveDancer']
    msg3 = [
        'jumpLeft',
        'leftLegUp',
        'clickHeels',
        'spreadOut',
        'spreadOut',
        'lineDancer',
        'disco',
        'leftLegUp',
        'spreadOut']
    msg=msg1 + msg2 + msg3
    letter_counts = dict()
    for symbol in msg:
        if symbol in letter_counts:
            letter_counts[symbol] += 1
        else:
            letter_counts[symbol] = 1
    for symbol in letter_counts.keys():
        print("symbol:", symbol, "count:", letter_counts[symbol])


dancingMen()
```

The dictionary named letter_counts begins as an empty dictionary, so the command is **letter_counts = dict()**. Each time a new symbol is encountered that symbol and a count of 1 are added to the dictionary by this command **letter_counts[symbol]** = 1. When the symbol has already been saved in the dictionary then the count is incremented by the command **letter_counts[symbol] += 1**, which is an abbreviation of **letter_counts[symbol] = letter_counts[symbol] = 1**.

The python program prints each symbol's name and its total below.



```
>>>
symbol: leftFootUp count: 1
symbol: clickHeels count: 2
symbol: disco count: 2
symbol: armsV count: 1
symbol: jumpLeft count: 1
symbol: irish count: 1
symbol: spreadOut count: 9
symbol: leftLegUp count: 3
symbol: lineDancer count: 3
symbol: raveDancer count: 1
symbol: leftLegUpRightArmOut count: 1
symbol: rightFootUp count: 4
symbol: rightLegUp count: 2
symbol: headStand count: 1
symbol: handStand count: 1
>>>
```

As Sherlock Holmes recalls the letter e is the most common letter in English, so the symbol named spreadOut is E. Read the story for the complete solution.

Challenge exercise: The current program does not list the symbols in order by count, so the user has to manually scan the totals. Enhance the display function to list the symbols and their counts from the largest to the smallest. Create an empty list. For each symbol in the dictionary create a tuple consisting of the symbol's name and count. Store that tuple in the list. Sort the list. Print the symbol counts from the list, not the dictionary.

Fun for all majors:

Step 1: Maintain a list of upcoming movie titles that you want to see.

Step 2: Instead of simply the title, use a tuple to store the movie's category, title, and release date)

Computer Science:

Count the occurrences of a letter within a string. This is a basic starting strategy for code cracking. The frequency of the letters, such as “e,” depends on the language.

Business:

Process a list of stock sales and purchases. Each sale is stored as a tuple containing the sale date stored as YYMMDD, the company's NYSE abbreviation, number of shares, and total price of the transaction. Ask the user which day and company to process. Display the total shares sold or purchased.

Software Engineering:

Maintain a list of the data sources for a project. The list hold tuples. Each tuple holds the abbreviated name of the data source, field names within each data source, data type, and description of each field.

Biotechnology:

DNA, described by Wikipedia at http://en.wikipedia.org/wiki/DNA, consists of a series of letters. Write a python function called chk_DNA_letters to check that a list of such DNA letters stored as strings contains only those appropriate DNA letters. Display only the good strings. Display a count of the number of good strings.

Mathematics:

Write an program to hold a finite series containing real numbers. Display a running total of the series as you proceed through the series.